使用Akka构建高容错、并发、分布式应用程序

Akka 入门与实践

Learning Akka

[加] Jason Goodwin 著
诸豪文 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

Akka入门与实践

[加] Jason Goodwin 著
诸豪文 译

人民邮电出版社
北京

图书在版编目（CIP）数据

Akka入门与实践 / (加) 贾森·古德温 (Jason Goodwin) 著 ; 诸豪文译. -- 北京 : 人民邮电出版社, 2017.6
ISBN 978-7-115-45354-9

Ⅰ. ①A… Ⅱ. ①贾… ②诸… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2017)第077808号

版权声明

◆ 著　[加] Jason Goodwin
译　诸豪文
责任编辑　王峰松
责任印制　马振武

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：14.5
字数：272 千字　2017 年 6 月第 1 版
印数：1－2 500 册　2017 年 6 月北京第 1 次印刷

著作权合同登记号　图字：01-2017-0521 号

定价：49.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第 8052 号

内容提要

本书主要面向使用 Akka 工具集来构建大规模分布式应用程序的 Java 和 Scala 开发者，介绍了分布式系统的基本概念以及如何使用 Akka 来构建容错性高、可横向扩展的分布式应用程序。书中主要内容包括：Akka 工具集、Actor 模型、响应式编程、Actor 及 Future 的使用、Akka 消息传递模式、Actor 生命周期、监督机制、状态与错误处理、Akka 并发编程、路由、阻塞 IO 的处理、Akka Cluster、CAP 理论、Akka 邮箱问题的处理、Akka Testkit、领域驱动设计等。本书贯穿使用了分布式键值存储以及文章解析服务两个实例，将原理与实践结合，介绍了使用 Akka 设计并实现分布式应用程序的方法。

关于作者

Jason Goodwin 是一个基本上通过自学成才的开发者。他颇具企业家精神，在学校学习商学。不过他从 15 岁起就开始学习编程，并且一直对技术保持着浓厚的兴趣。这对他的职业生涯产生了重要的影响，从商学转向了软件开发。现在他主要从事大规模分布式系统的开发。在业余时间，他喜欢自己原创电子音乐。

他在 mDialog 公司第一次接触到 Akka 项目。mDialog 是一家使用 Scala/Akka 的公司，为主流出版商提供视频广告插入软件。这家公司最终被 Google 收购。他同时还是一名很有影响力的“技术控”，将 Akka 引入加拿大一家主要的电信公司，帮助该公司为客户提供容错性更高、响应更及时的软件。除此之外，他还为该公司中的一些团队教授 Akka、函数式以及并发编程等知识。

关于译者

诸豪文，网名 clasnake，毕业于清华大学，现为全职软件开发工程师，常用的开发语言有 Java、Scala、JavaScript 和 Python。其个人博客地址为 http://clasnake.net。他也是开源项目 Swagger 的贡献者，并译有《Python 网络编程》（第 3 版）一书。

技术审核

Taylor Jones 是一名全栈软件工程师，精通基于 Java 的 Web 应用程序开发，目前在 Cisco Systems 工作。他很喜欢使用开源技术设计并构建复杂的应用程序。

致谢

我在这里想要感谢一些人，你们对我的价值观的形成有着很大的影响，并且给予我不断的支持。

首先，要感谢我的妻子 Kate，谢谢你在我撰写本书期间以及做一些疯狂项目的时候对我的支持。如果没有你不断的支持、耐心和照顾，我就不可能改变我的职业生涯去做我喜欢的事情，也不可能写出来这本书。我们终于完成了。现在是时候去粉刷一下墙壁，修缮一下屋子，看着电影休息一下了！

感谢我的祖父母和父母，你们一直告诉我，只要下定了决心，什么事都难不倒我。谢谢你们的建议。你们是对的。

感谢我的 mDialog/Google 团队，感谢你们的审阅和批评。能有机会和你们共事，我觉得非常幸运。特别要感谢 Chris，感谢你相信我的兴趣足以帮助我成长为一个合格的工程师，也感谢你一直都希望团队能够给予我支持。

感谢 Graig 和 Herb，感谢你们对我的启蒙。如果我没有在 17 岁的时候编写冒泡排序、画像素圆或是移植客户数据库，那么我自己都不确定是不是能像现在一样找到自己这么热爱的工作。

Jason Goodwin

前言

本书将尝试帮助入门级、中级以及高级读者理解基本的分布式计算概念，并且展示如何使用 Akka 来构建具备高容错性、可以横向扩展的分布式网络应用程序。Akka 是一个强大的工具集，提供了很多选项，可以对在本地机器上处理或网络远程机器上处理的某项工作进行抽象封装，使之对开发者不可见。本书将介绍各种概念，帮助读者理解网络上各系统进行交互的困难之处，并介绍如何使用 Akka 提供的解决方案来解决这些问题。

编写本书的过程也是作者自我学习、自我发现的过程。希望读者也能一起分享这些知识。作者在工作中有很多使用 Java 8 和 Scala 来编写 Akka 应用程序的经验，但是在编写本书的过程中，学到了很多更深入的 Akka 细节。这本书很好地介绍了为什么要使用 Akka，以及如何使用 Akka，并且展示了如何使用 Akka 工具集来开始构建可扩展的分布式应用程序。本书并不仅仅是官方文档的重复，更涉及了许多作为当代程序员要成功构建能够处理扩展性相关问题的系统时应该要理解的重要话题和方法。

本书涉及的内容

第 1 章　初识 Actor：Akka 工具集以及 Actor 模型的介绍。

第 2 章　Actor 与并发：响应式编程。Actor 与 Future 的使用。

第 3 章　传递消息：消息传递模式。

第 4 章　Actor 的生命周期——处理状态与错误：Actor 生命周期、监督机制、Stash/Unstash、Become/Unbecome 以及有限自动机。

第 5 章　纵向扩展：并发编程、Router Group/Pool、Dispatcher、阻塞 I/O 的处理以及 API。

第 6 章　横向扩展——集群化：集群、CAP 理论以及 Akka Cluster。

第 7 章　处理邮箱问题：加大邮箱负载、不同邮箱的选择、熔断机制。

第 8 章　测试与设计：行为说明、领域驱动设计以及 Akka Testkit。

第 9 章　尾声：其他 Akka 特性。下一步需要学习的知识。

阅读本书的前提条件

读者需要一台能够安装各种工具的计算机，比如 Java JDK8（用于 Java 开发）或是 Java JDK6（用于 Scala 开发）。除此之外，还需要 SBT 简单构建工具（Simple Build Tool）或是 Typesafe Activator（已经包含 SBT）。本书会介绍这些工具的安装。

本书的目标读者

本书面向想要构建满足大规模用户需求的应用程序的初级至中级 Java 或 Scala 开发者。如果应用程序在处理日益增加的用户量以及数据量时需要满足高性能要求，那么建议阅读本书。本书可以让我们只编写更少、更简单的代码就轻松构建并扩展网络应用程序，给用户提供更强大的功能。

读者反馈

我们欢迎读者的反馈。对本书的任何看法敬请告知我们。读者反馈对我们至关重要，可以帮助我们出版读者真正能够从中获益的书籍。

如果有普通的反馈，请直接向 feedback@packtpub.com 发送电子邮件，并且在邮件标题中提及书名。

如果有您精通的话题并且有兴趣撰写书籍或是向某本书做贡献，请从 www.packtpub.com/authors 查阅作者手册。

客户支持

您是 Packt 书籍的拥有者，所以我们提供了很多服务，帮助您从所购买的书中获得最大的收获。

下载示例代码

读者可以使用自己的账户从 http://www.packtpub.com 下载购买过的所有 Packt Publishing 书籍的示例代码文件。如果从别处购买了本书，那么可以访问 http://www.packtpub.com/support 并进行注册，我们会通过电子邮件将文件发送给您。

下载本书的彩色图片

我们还提供了一个 PDF 文件，其中包含本书中使用的截屏/图表的彩色图片。这些彩色图片可以帮助读者更好地理解输出结果中的不同之处。读者可以从 https://www.packtpub.com/sites/default/files/downloads/LearningAkka_ColoredImages.pdf 处下载该文件。

电子书、折扣以及其他

Packt 出版社为出版的每一本书都提供了 PDF 和 ePub 的电子书版本。读者可以从 www.packtpub.com 获取电子书版本。纸质书的客户在购买电子书时可以享受折扣优惠。请通过 customercare@packtpub.com 联系我们，获取更多细节。

您还可以在 www.packtpub.com 阅读更多免费的技术文章，订阅免费新闻，并接收到大量 Packt 书籍以及电子书的折扣优惠。

问题

如果对本书有任何问题，请通过 questions@packtpub.com 联系我们，我们将尽最大的努力解决您的问题。

目录

第 1 章　初识 Actor 1

1.1　本章概述 1

1.2　什么是 Akka 1

1.2.1　Actor 模型的起源 1

1.2.2　什么是 Actor 2

1.2.3　Actor 和消息传递 2

1.3　本书示例系统 7

1.3.1　示例 1：处理分布式状态 7

1.3.2　示例 2：完成更多工作 8

1.4　配置环境 8

1.4.1　选择一门语言 9

1.4.2　安装 Java——Oracle JDK8 9

1.4.3　确认 Java 环境配置 10

1.4.4　安装 Scala 10

1.4.5　安装 Typesafe Activator 10

1.4.6　新建项目 11

1.4.7　安装 IDE 12

1.5　创建第一个 Akka 应用程序——设置 SBT 项目 15

1.5.1　将 Akka 添加至 build.sbt 16

1.5.2　创建第一个 Actor 17

1.5.3　使用单元测试验证代码 21

1.5.4　运行测试用例 24

1.6　课后作业 25

1.7　小结 26

第 2 章　Actor 与并发 27

2.1　响应式系统设计 27

2.2 响应式四准则……28
2.2.1 灵敏性……28
2.2.2 伸缩性……28
2.2.3 容错性……28
2.2.4 事件驱动/消息驱动……28
2.2.5 响应式准则的相关性……29
2.3 剖析 Actor……29
2.3.1 Java Actor API……29
2.3.2 Scala Actor API……32
2.4 Actor 的创建……33
2.5 Promise、Future 和事件驱动的编程模型……36
2.5.1 阻塞与事件驱动 API……36
2.5.2 使用 Future 进行响应的 Actor……40
2.5.3 理解 Future 和 Promise……45
2.5.4 在失败情况下执行代码……49
2.5.5 从失败中恢复……49
2.5.6 异步地从失败中恢复……50
2.5.7 链式操作……51
2.5.8 组合 Future……51
2.5.9 处理 Future 列表……52
2.5.10 Future 速查表……53
2.5.11 准备数据库与消息……54
2.5.12 编写客户端……59
2.6 课后作业……62
2.6.1 基本知识……62
2.6.2 项目作业……62
2.7 小结……63

第 3 章 传递消息 64

3.1 示例问题……64
3.2 消息传递……65
3.2.1 消息是不可变的……66
3.2.2 Ask 消息模式……69

3.2.3 Tell ····· 78
3.3 课后作业 ····· 88
3.4 小结 ····· 88

第 4 章 Actor 的生命周期——处理状态与错误 90

4.1 分布式计算的 8 个误区 ····· 90
4.1.1 网络是可靠的 ····· 90
4.1.2 没有延迟 ····· 91
4.1.3 带宽是无限的 ····· 91
4.1.4 网络是安全的 ····· 92
4.1.5 网络拓扑不会改变 ····· 92
4.1.6 只有一个管理员 ····· 92
4.1.7 网络传输没有开销 ····· 93
4.1.8 网络是同构的 ····· 93
4.2 错误 ····· 93
4.2.1 隔离错误 ····· 94
4.2.2 监督 ····· 95
4.3 状态 ····· 102
4.3.1 在线/离线状态 ····· 103
4.3.2 条件语句 ····· 104
4.3.3 热交换（Hotswap）：Become/Unbecome ····· 105
4.3.4 通过重启转移状态 ····· 113
4.4 课后作业 ····· 113
4.5 小结 ····· 114

第 5 章 纵向扩展 115

5.1 摩尔定律 ····· 115
5.2 多核架构的分布式问题 ····· 116
5.3 选择 Future 或 Actor 进行并发编程 ····· 117
5.4 并行编程 ····· 117
5.4.1 使用 Future 进行并行编程 ····· 118
5.4.2 使用 Actor 进行并行编程 ····· 119
5.5 使用 Dispatcher ····· 123

5.5.1 Dispatcher 解析 ····· 123
5.5.2 Executor ····· 124
5.5.3 创建 Dispatcher ····· 124
5.5.4 决定何时使用哪种 Dispatcher ····· 126
5.5.5 默认 Dispatcher ····· 128
5.5.6 使用 Future 的阻塞 IO Dispatcher ····· 130
5.5.7 用于解析文章的 Dispatcher ····· 132
5.5.8 并行最优化 ····· 135
5.6 **课后作业** ····· 135
5.7 **小结** ····· 136

第 6 章 横向扩展——集群化 137

6.1 **Akka Cluster 介绍** ····· 137
6.2 **巨型单体应用 vs 微服务** ····· 137
6.3 **集群的定义** ····· 138
6.3.1 失败检测 ····· 139
6.3.2 通过 gossip 协议达到最终一致性 ····· 139
6.4 **CAP 理论** ····· 140
6.4.1 C –一致性（Consistency） ····· 140
6.4.2 A –可用性（Availability） ····· 140
6.4.3 P –分区容错性（Partition Tolerance） ····· 140
6.4.4 CAP 理论中的妥协 ····· 141
6.5 **使用 Akka Cluster 构建系统** ····· 143
6.5.1 创建集群 ····· 143
6.5.2 集群成员的状态 ····· 150
6.5.3 通过路由向集群发送消息 ····· 151
6.5.4 编写分布式文章解析服务 ····· 151
6.5.5 用于集群服务的集群客户端 ····· 153
6.5.6 集群设计 ····· 159
6.6 **结合分区与冗余** ····· 164
6.7 **远程 Actor 寻址** ····· 166
6.8 **课后作业** ····· 167
6.9 **小结** ····· 167

第 7 章　处理邮箱问题 169

7.1　搞垮最可能出问题的服务……169
7.1.1　响应时间变长……170
7.1.2　崩溃……171
7.2　恢复能力……171
7.3　在高负载情况下保持响应速度……175
7.4　课后作业……181
7.5　小结……182

第 8 章　测试与设计 183

8.1　示例问题……183
8.2　应用程序设计……184
8.3　设计、构建并测试领域模型……186
8.3.1　行为说明……186
8.3.2　设计领域模型……187
8.3.3　构建并测试领域模型……188
8.3.4　基于行为说明编写代码……190
8.4　测试 Actor……192
8.4.1　测试 Actor 行为及状态……192
8.4.2　测试消息流……195
8.5　测试建议……198
8.6　课后作业……199
8.7　小结……200

第 9 章　尾声 201

9.1　其他 Akka 功能及模块……201
9.1.1　Akka 中的日志……202
9.1.2　消息信道与 EventBus……204
9.1.3　Agent……206
9.1.4　Akka Persistence……209
9.1.5　Akka I/O……210
9.1.6　Akka Streams 与 HTTP……210

9.2 部署工具……210
9.3 监控日志与事件……212
9.4 下一步……212
9.4.1 编写一些 Actor 代码……213
9.4.2 Coursera 课程……213
9.5 小结……214

第 1 章
初识 Actor

1.1 本章概述

Actor 模型是一种并发计算的理论模型，而 Akka 的核心其实是 Actor 模型的一种实现。在本章中，我们将通过了解 Akka 和 Actor 模型的历史来介绍 Akka 的核心概念。这会帮助读者更好地理解 Akka 到底是什么，以及 Akka 试图要解决什么样的问题。其次，本章中将重复使用同一个例子来阐述本书的目的。

在介绍了上面这些概念后，本章将会把篇幅放在开发环境及工具的配置方法上。我们将配置好机器的环境，集成开发环境（Integrated Development Environment，IDE）并介绍第一个 Akka 项目（包括该项目的单元测试）。

1.2 什么是 Akka

本节将介绍 Akka 和 Actor 模型。Akka 一词据说来源于瑞典的一座山，我们说到 Akka 时，通常是指一个分布式工具集，用于协调远程计算资源来进行一些工作。Akka 是 Actor 并发模型的一种现代化实现。现在的 Akka 可以认为是从许多其他技术发展演化而来的，它借鉴了 Erlang 的 Actor 模型实现，同时又引入了许多新特性，帮助构建能够处理如今大规模问题的应用程序。

1.2.1 Actor 模型的起源

为了更好地理解 Akka 的含义及其使用方法，我们将快速地了解 Actor 模型的历史，理解 Actor 模型的含义，以及它是如何一步一步发展到如今的 Akka 这样一个用于构建高容错性分布式系统的框架。

Actor 并发模型最早出现于一篇叫作《A Universal Modular Actor Formalism for Artificial Intelligence》的论文，该论文发表于 1973 年，提出了一种并发计算的理论模型，Actor 就源于该模型。我们将在本节中学习 Actor 模型的特性，理解它的优点，能够在并发计算中帮助我们解决共享状态带来的常见问题。

1.2.2　什么是 Actor

首先，让我们来定义什么是 Actor。在 Actor 模型中，Actor 是一个并发原语；更简单地说，可以把一个 Actor 看作是一个工人，就像能够工作或是处理任务的进程和线程一样。把 Actor 看成是某个机构中拥有特定职位及职责的员工可能会对理解有所帮助。比如说一个寿司餐馆。餐馆的职员需要做各种各样不同的工作，给客人准备餐盘就是其中之一。

1.2.3　Actor 和消息传递

在面向对象编程语言中，对象的特点之一就是能够被直接调用：一个对象可以访问或修改另一个对象的属性，也可以直接调用另一个对象的方法。这在只有一个线程进行这些操作时是没有问题的，但是如果多个线程同时读取并修改同一个值，那么可能就需要进行同步并加锁。

Actor 和对象的不同之处在于其不能被直接读取、修改或是调用。反之，Actor 只能通过消息传递的方式与外界进行通信。简单来说，消息传递指的是一个 Actor 可以接收消息（在我们的例子中该消息是一个对象），本身可以发送消息，也可以对接收到的消息作出回复。尽管我们可以将这种方式与向某个方法传递参数并接收返回值进行类比，但是消息传递与方法调用在本质上是不同的：消息传递是异步的。无论是处理消息还是回复消息，Actor 对外界都没有依赖。

Actor 每次只同步处理一个消息。邮箱本质上是等待 Actor 处理的一个工作队列，如图 1-1 所示。处理一个消息时，为了能够做出响应，Actor 可以修改内部状态，创建更多 Actor 或是将消息发送给其他 Actor。

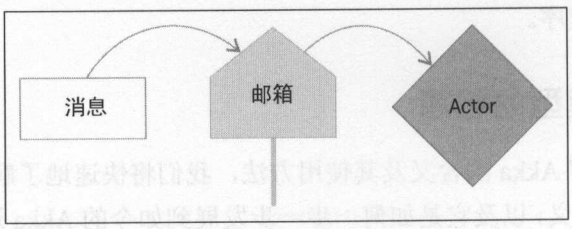


图 1-1

在具体实现中，我们通常使用 Actor 系统这个术语来表示多个 Actor 的集合以及所有与该 Actor 集合相关的东西，包括地址、邮箱以及配置。

下面再重申一下这几个重要的概念：

- **Actor**：一个表示工作节点的并发原语，同步处理接收到的消息。Actor 可以保存并修改内部状态。
- **消息**：用于跨进程（比如多个 Actor 之间）通信的数据。
- **消息传递**：一种软件开发范式，通过传递消息来触发各种行为，而不是直接触发行为。
- **邮箱地址**：消息传递的目标地址，当 Actor 空闲时会从该地址获取消息进行处理。
- **邮箱**：在 Actor 处理消息前具体存储消息的地方。可以将其看作是一个消息队列。
- **Actor 系统**：多个 Actor 的集合以及这些 Actor 的邮箱地址、邮箱和配置等。

虽然现在看来可能还不是太明显，但是 Actor 模型要比命令式的面向对象并发应用程序容易理解多了。我们可以举一个现实世界中的例子来比喻使用 Actor 模型来建模的过程，这会帮助我们理解它带来的好处。比如有一个寿司餐馆，其中有 3 个 Actor：客人、服务员以及厨师。

首先，客人向服务员点单。服务员将客人点的菜品写在一张纸条上，然后将这张纸条放在厨师的邮箱中（将纸条贴在厨房的窗户上）。当厨师有空闲的时候，就会获取这条消息（客人点的菜品），然后就开始制作寿司（处理消息），直至寿司制作完成。寿司准备好以后，厨师会发送一条消息（盛放寿司的盘子）到服务员的邮箱（厨房的窗户），等待服务员来获取这条消息。此时厨师可以去处理其他客人的订单。

当服务员有空闲时，就可以从厨房的窗户获取食物的消息（盛放寿司的盘子），然后将其送到客人的邮箱（比如餐桌）。当客人准备好的时候，他们就会处理消息（吃寿司），如图 1-2 所示。

运用餐厅的运作来理解 Actor 模型是很容易的。随着越来越多的客人来到餐厅，我们可以想象服务员每次接收一位客人的下单，并将订单交给厨房，接着厨师处理订单制作寿司，最后服务员将寿司交给客人。每个任务都可以并发进行。这就是 Actor 模型提供的最大好处之一：当每个人各司其职时，使用 Actor 模型分析并发事件非常容易。而使用 Actor 模型对真实应用程序的建模过程和本例中对寿司餐厅的建模过程并没有太大差异。

Actor 模型的另一个好处就是可以消除共享状态。因为一个 Actor 每次只处理一条消息，所以可以在 Actor 内部安全地保存状态。如果读者此前没有接触过并发系统，那么可能不是很容易马上理解这一点。不过我们可以用一种简单的方式来进行说明。假设我们尝试执行两个操作，同时读取、修改并保存一个变量，那么如果我们不进行同步操作并加锁的话，其中的一个操作结果将丢失。这是一个非常容易犯的错误。

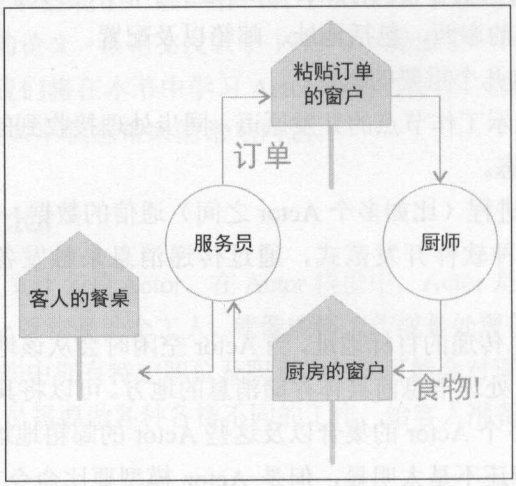


图 1-2

在下面的例子中，有两个线程同时执行一个非原子的自增操作。让我们来看看在线程间共享状态会带来什么结果。会有多个线程从内存中读取一个变量值，将该变量自增，然后将结果写回内存。这就是竞态条件（Race Condition），可以通过保证某一时刻只有一个线程访问内存中的值来解决其带来的一部分问题。下面用一个 Scala 的例子进行说明。

如果我们尝试使用多个线程并发地对一个整型变量执行 100 000 次自增操作，那么极有可能会丢失掉其中一些自增操作的结果。

```
import concurrent.Future
import concurrent.ExecutionContext.Implicits.global
var i, j = 0
(1 to 100000).foreach(_ => Future{i = i + 1})
(1 to 100000).foreach(_ => j = j + 1)
Thread.sleep(1000)
println(s"${i} ${j}")
```

我们使用 $x = x + 1$ 这个简单的函数将 i 和 j 都自增了 100 000 次，其中 i 的自增操作通过多个线程并发执行，而 j 的自增操作则只通过一个线程来执行。等待 1 秒钟后，我们再打印运行结果，确保所有更新都已经完成。读者可能会认为运行结果是 100000 100000，然而结果却并非如此，如图 1-3 所示。

共享状态是不安全的。如果两个线程同时读取一个值，将该值自增，然后写回内存，那么由于该值同时被多个线程读取，其中的某些操作结果将会丢失。这就是竞态条件，也是使用共享状态的并发模型存在的最基本的问题之一。

```
scala> (1 to 100000).foreach(_ => Future{i = i + 1})

scala> (1 to 100000).foreach(_ => j = j + 1)

scala> println(s"${i} ${j}")
71914 100000
```

图 1-3

通过推导每一步读取和写入操作的结果，我们能够更清晰地展示出竞态条件发生时的具体情况：

```
[...]
Thread 2 reads value in memory - value read as 9
Thread 2 writes value in memory - value set to 10 (9 + 1)
Thread 1 reads value in memory - value read as 10
Thread 2 reads value in memory - value read as 10
Thread 1 writes value in memory - value set to 11 (10 + 1) !! LOST
INCREMENT !!
Thread 2 writes value in memory - value set to 11 (10 + 1)
Thread 1 reads value in memory - value read as 11
[...]
```

为了保证内存中的共享状态值不出现错误，我们可以使用锁和同步机制，防止多个线程同时读取并写入同一个值。这就会导致问题变得更为复杂，更难理解，也更难确保结果的正确。

使用共享状态带来的最大威胁就是代码在测试中看上去经常是正确的，但是一旦有多个线程并发运行时，就会时不时地出现一些错误。由于测试时通常都不会出现多线程并发的情况，因此这些 Bug 很容易被忽略。Dion Almaer 曾经在博客中写到过，大多数 Java 应用程序都存在大量的并发 Bug，因此有时能正确运行，有时却运行失败。Actor 通过减少共享状态来解决这一问题。如果我们把状态移到 Actor 内部，那么只有该 Actor 才能访问其内部的状态（实际上只有一个线程能够访问这些内部状态）。如果把所有消息都看做是不可变的，那么我们实际上可以去除 Actor 系统中的所有共享状态，构建更安全的应用程序。

本小节中的概念是 Actor 模型中的核心。第 2 章 Actor 与并发和第 3 章发送消息将会更详细地介绍并发、Actor 以及消息传递。

Erlang 语言中监督和容错机制的发展演化

自从在前面提到的论文中第一次出现以来，Actor 模型随着时间的推移不断地发展，它对程序语言设计也产生了显著影响（比如 Scheme）。

20 世纪 80 年代，爱立信在 Erlang 语言中实现了 Actor 模型，用于嵌入式电信应用程序。这一实现绝对值得一提。该实现中引入了通过监督机制（Supervision）提供的容错性概念。爱立信使用 Erlang 和 Actor 模型实现了一款日后经常被提及的应用，叫作 AXD301。AXD301 能够提供 99.9999999%的可用性，这一点令人惊叹。相当于在 100 年中，AXD301 只有 3.1 秒的时间会宕机。AXD 的开发团队表示，他们通过消除共享状态（正如我们之前介绍的一样）并引入 Erlang 中的监督容错机制来达到如此高的可用性。

Actor 模型也是通过监督机制来提供容错性的。监督机制基本上是指把处理响应错误的责任交给出错对象以外的实体。这意味着一个 Actor 可以负责监督它的子 Actor，它会监控子 Actor 的运行错误，并且根据子 Actor 生命周期中的运行表现执行相应的操作。当一个正在运行的 Actor 发生错误时，监督机制提供的默认处理方式是重新启动发生错误的 Actor（实际上是重新创建）。这种重新创建出错 Actor 的处理方式基于一种假设：意外发生的错误是由错误状态导致的，因此移除并重新创建应用程序中出错的部分可以将其修复，并恢复正常工作。我们也可以编写自定义的错误处理方式作为监督策略，这样一来基本上就可以采取任何操作将应用程序恢复至工作状态，如图 1-4 所示。

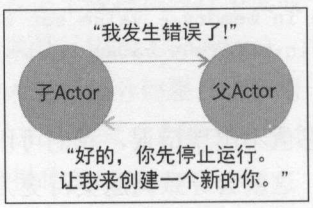


图 1-4

在本书中，我们会将关于分布式系统的容错性作为一个通用的问题，在多个章节中交叉介绍，并且把重点放在 Akka 和分布式系统的容错性上。第 4 章“Actor 的生命周期——处理状态与错误”将重点介绍这一内容。

分布式与位置透明性的发展演化

如今的业务需求要求工程师能够设计同时响应成千上万个并发用户请求的系统，而用一台机子来运行这样的系统是绝对不够的。除此之外，多核处理器变得越来越流行，因此将任务分布到多个核上以确保有效地利用硬件资源也变得越来越重要。

Akka 采用了 Actor 模型，并且继续对其发展演化，引入了对如今的工程师来说最重要的一个特性：网络环境下的分布式。Akka 将其自己视作是一个支持容错性的分布式工具集。也就是说，Akka 是一个提供了在多个服务器的物理边界之间工作的工具集。在支持高可用性的同时，几乎可以无限扩展。在最近的几个 Akka 发布版本中，大多数新增的特性都和解决网络系统的问题有关。最近引入的 Akka Cluster（集群）允许将一个 Actor 系

统部署到多台机器上，并且这一点对用户不可见。Akka IO 和 Akka HTTP 也已经进入了核心库，使得系统之间的交互变得更简单。Akka 对于 Actor 模型的重要贡献之一就是位置透明性的概念：就是说一个 Actor 的邮箱地址实际上可以是一个远程地址，但是这个地址对开发者来说基本上是透明的，所以无论是否是远程地址，编写的代码也基本上是相同的。

Akka 沿用了 Erlang 的一些 Actor 实现，并且打破了 Actor 系统的物理边界。Akka 添加了远程处理以及位置透明性，使得一个 Actor 的邮箱可以在远程机器上，而 Akka 会对网络上的消息传输进行抽象封装。

最近，Akka 又引入了集群。读者可能知道一些基于 Amazon 那篇 Dynamo 论文的分布式系统，比如 Dynamo、Cassandra 和 Riak。Akka Cluster 也采用了类似的现代化方法。有了 Cluster 以后，一个 Actor 系统就可以运行在多台机器之上，而不同的节点之间也可以就各自的状态互相通信交互，这就实现了一个可伸缩的 Akka 集群，并且没有单点故障。这一机制和 Dynamo 风格的数据库类似，比如 Riak 和 Cassandra。这个特性非常好，使得创建可伸缩并且具备优良容错性的系统变得相当容易。

Typesafe（提供 Scala 和 Akka 等技术的公司）[①]不断在通过提供许多网络工具（比如 Akka IO 和 Akka HTTP）来推广分布式计算。除此之外，Typesafe 已经参与了 Reactive Streams 提案，而 Akka 也实现了第一个版本，支持用于异步处理的非阻塞背压（Back-Pressure）。

在本书中，我们会详细介绍上面诸多内容。第 4 章 Actor 的生命周期——处理状态与错误和第 5 章纵向扩展将更详细地介绍远程处理。第 6 章横向扩展——集群化将介绍 Cluster。第 7 章处理邮箱问题将介绍 Reactive Streams。

1.3 本书示例系统

在本书中，我们将主要构建两个服务，推荐读者着手操作，一起一步一步地实践。在每章的结尾都有一些课后作业，通过这些练习，读者可以将这一章学到的内容付诸实践。在开始下一章的学习之前，请完成这些课后作业。如果想和别人分享学习的进度，或是希望将这些作业的答案开源，那么可以把它们发布到 GitHub 上。

在本书中，我们将主要开发两个软件。第一个例子用于展示状态和分布式的处理；第二个例子用于展示如何完成工作。

1.3.1 示例 1：处理分布式状态

我们将研究如何构建一个可扩展的分布式内存数据库。在另一个例子中，我们将把数据存储到这个数据库中。说得明确一点，我们将构建一个类似于 Redis 或 Memcached 的高

① 译者注：已改名为 Lightbend。

可用键值存储。我们构建的数据库将处理其在真实应用场景下所需的所有并发、集群以及分布式问题。为了了解如何提供上面的诸多特性，我们应学习如何在集群中对数据库的数据和负载进行切分及分配，有效地利用硬件资源以及如何进行横向扩展利用多台机器。我们将了解面对现实问题时的设计挑战及常见解决方案。我们还将研究如何构建客户端库，与我们编写的基于 Akka 的数据库进行交互，并且允许 JVM 上的任意用户使用。我极力推荐读者自己编写一个这样的数据库，放到 GitHub 上面去，并且在简历里展示一下。

这看上去相当复杂，但是庆幸的是，有了 Akka 工具集，要完成上面所有的任务其实相当简单。我们将从零开始，很快地构建出这个完整的系统。

1.3.2　示例 2：完成更多工作

为了在本书中举例展示如何使用 Akka 完成更多的工作，我们将构建一个用于阅读文章的 API，读取博客或是新闻文章，抽取出主要的文本内容，然后将其存储到我们编写的数据库中，作为其他应用的数据源。

下面是一个用例：假设有一个移动设备上的阅读器，通过我们编写的服务从流行的 RSS Feed 请求读取文章，然后显示主要的文本内容，能够缩放文字适应屏幕显示，提供更好的阅读体验。我们的服务负责从主要的 RSS Feed 解析出文本内容，使其在移动设备上显示速度更快，用户无需等待。如果想要体验这样的真实移动应用程序，可以看看 iOS 上的 Flipboard，这是我们所编写服务的应用的一个很好的例子，如图 1-5 所示。

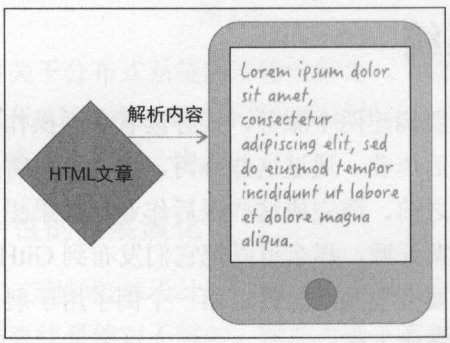


图 1-5

我们已经介绍了本书涉及的内容，接下来就开始配置环境并创建一个 Actor 吧！

1.4　配置环境

在真正深入学习 Akka 之前，我们将介绍开发环境以及项目基本结构的设置。在本

书后面的章节中新建项目时，读者可以回过头来参考本节的内容。

1.4.1 选择一门语言

Scala 和 Java 的 API 基本上是一一对应的，所以读者选择自己熟悉的语言即可。如果既会 Scala，又会 Java，那么 Scala API 当然更符合推荐的风格。不过两者都是绝对可用的选择。因为我们可以通过 Scala 的 Actor API 使用 Scala 来访问使用 Java 构建的 Actor，反之亦然，所以也没有必要立刻就做出决定。选择能够让自己更快上手开发的就行。现在我们的重点是学习 Akka，而不是学习语言。一旦了解了 Akka，要学习另一个语言的 API 并不需要费很大的功夫。

1.4.2 安装 Java——Oracle JDK8

本书并不支持所有旧版本的 Java，只支持 Java 8。因此如果读者是一个 Java 开发者，但是对 Java 8 的特性并不熟悉，那么应该花点时间了解一下 lambda 和 stream API。下面的教程对此进行了介绍：

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html。

读者将会发现，本书中大量使用了 lambda。因此花时间了解一下绝对是有帮助的。

在 Windows 上安装

从 Oracle 下载并安装 Windows JDK8 的安装包（exe）：http://www.oracle.com/technetwork/java/javase/downloads/index.html。

按照说明安装。

在 OS X 上安装

从 Oracle 下载并安装 OS X JDK8 的安装包（dmg）：http://www.oracle.com/technetwork/java/javase/downloads/index.html。

按照说明安装。

在 Linux 或 Unix 上安装（通用步骤）

有很多方法可以用于在*Nix 系统上安装 Java。我们可以使用通用安装包，也可以使用包管理器，比如基于 Red Hat Enterprise Linux（RHEL）的发行版上的 Yum 和基于 Debian 的发行版上的 Apt-Get。不同发行版上包管理器的安装方法各不相同，不过如果需要的话，可以在 Google 搜索引擎上找到相应的安装步骤。

通用安装包适用于所有系统，因此我们在这里介绍这种方法。这也是能够使用的最

基本的安装方法。它会安装 JDK，并且为当前用户启动 JDK，但是不会对系统做出更改。如果想要修改系统的 JDK/JRE，可以参照所运行的特定发行版的安装步骤。这一点对于服务器和桌面环境都是适用的。如果使用的是桌面环境，可以参照一下为其他用户设置默认 JDK/JRE 的步骤。

从 Oracle 下载 Linux 的 JDK 安装包（tar.gz）：http://www.oracle.com/ technetwork/java/javase/downloads/index.html

该文件的名字可能类似 jdk-8u31-linux-x64.tar.gz。将 tar.gz 的文件解压到合适的位置，比如/opt：

```
sudo cp jdk-8u31-linux-x64.tar.gz /opt cd /opt sudo tar -xvf jdk-
8u31-linux-x64.tar.gz
```

我们需要将 JAVA_HOME 设置为 Java8 的文件夹：

```
echo 'export JAVA_HOME=/opt/jdk1.8.031' >> ~/.profile
```

并且确认 PATH 包含 Java 的 bin 目录：

```
echo 'export PATH=$PATH:/opt/jdk1.8.031' >> ~/.profile
```

现在，IDE 和 Sbt/Activator 就可以使用安装的 JDK 来生成并运行我们所构建的应用程序了。

1.4.3　确认 Java 环境配置

无论运行的是哪个操作系统，我们都需要确认设置了 JAVA_HOME，并且 PATH 包含了 Java 的 bin 目录。只有使用通用安装包的时候才需要手动设置 JAVA_HOME 和 PATH，但是无论使用哪种安装方式，都应该在一个新建的终端窗口中确认配置了 JAVA_HOME 环境变量并向 PATH 中添加了 Java 的 bin 文件夹。

1.4.4　安装 Scala

如果使用的是 Scala，那么需要在系统上安装 Scala 及其 REPL。在本书撰写时，最新的 Scala 版本（2.11）会被编译成 Java 1.6 的字节码，因此无需安装 JDK8。不过有人说未来的 Scala 版本可能会要求 JDK8，所以这一点可能会有变化。

Scala 并不一定要单独安装。Typesafe Activator 包含了 Scala 以及我们需要的所有工具，接下来我们会进行安装。

1.4.5　安装 Typesafe Activator

Typesafe Activator 的安装包内打包了 Scala、Akka、Play，简单构建工具（Simple Build

Tool，SBT）以及其他一些特性，比如项目结构与模板。

Windows

从 Typesafe 下载 Typesafe Activator：http://www.typesafe.com/get-started

运行安装程序，按照屏幕上的步骤安装。

Linux/Unix/OS X

从 Typesafe 下载 Typesafe Activator：

- http://www.typesafe.com/activator/download。
- http://www.typesafe.com/get-started。

将文件解压到合适的位置，比如/opt：cd /opt sudo unzip typesafe- activator-1.2.12.zip。

赋予 Activator 可执行权限：sudo chmod 755 /opt/activator-1.2.12/activator。

将 Activator 添加至 PATH：echo 'export PATH=$PATH:/opt/activator- 1.2.12'。

退出并重新登录。确认可以在命令行运行下述命令：

```
activator --version
```

这句命令会输出类似如下的结果：sbt launcher version 0.13.5

OS X

除了可以使用 Linux 上的方法安装 Activator 外，还可以使用 brew。本小节介绍使用 brew 的安装方法。

打开一个终端窗口。

输入下述命令（拷贝自 http://brew.sh）。该命令会安装 Homebrew OS X 包管理器。

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
install/master/install)"
```

最后，在终端窗口中输入下述命令并按回车键：

```
brew install typesafe-activator
```

确认可以从命令行访问 Activator：

```
activator --version
```

1.4.6 新建项目

在本书中，我们将使用 Activator 快速建立项目的结构。我们可以使用任意模板来生成一个项目。我们只会使用基本的 Java 和 Scala 模板。读者可以随意尝试其他选

项。Typesafe 有许多由用户提交的模板，这些模板展示了如何结合使用各种不同的技术和方法。

在终端窗口的命令行提示符中输入如下命令，创建一个新的 Activator 模板：

```
activator new
```

该命令的输出如下。

从中选择一个模板或者键入模板的名称：

- minimal-akka-java-seed
- minimal-akka-scala-seed
- minimal-java
- minimal-scala
- play-java
- play-scala

提示

可以敲击 Tab 键查看所有模板

根据想要使用的语言，选择 minimal-scala 或者 minimal-java。接着会要求输入应用程序的名称，可以叫作 akkademy-db。

进入 akkademy-db 文件夹，运行 activator test，确认项目及环境已正确配置。

```
cd akkademy-db activator test
```

在输出中可以看到，项目进行了编译，并且运行了测试。如果有任何问题，读者可能需要在继续往下阅读前查询 stack-overflow，解决环境配置问题。

如果没有任何问题的话，可以看到下述输出：

```
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1 [success] Total
time: 3 s, completed 22-Jan-2015 9:44:21 PM
```

1.4.7　安装 IDE

我们已经配置好了环境，并且可以运行。现在可以开始编写代码了。如果读者想使用简单的文本编辑器，那么请跳过本小节。Emacs 和 Sublime 都是非常优秀的文本编辑器，提供语法高亮和支持自动补全的插件。不过如果读者想要使用 IDE 的话，本小节会介绍 Eclipse 和 Intellij 的配置。

安装 Intellij CE

如果选择使用 IDE 的话，推荐 Intellij。在撰写本书的过程中，和我共事的许多 Java 开发者接手开发 SBT 项目后，他们几乎全部都转而使用 Intellij，并且再也不想改回去了。

Intellij 现在有内置的 SBT 支持，因此用于开发 Akka 项目的时候速度会很快。由于 Intellij 原生地支持本书中用到的所有技术，因此用户几乎不需要进行任何 IDE 配置。

创建并运行项目的步骤：

1. 下载并安装 Intellij CE（免费）。
2. 安装完成后，选择打开项目。选择 akkademy-db 文件夹。
3. 如果使用的是 Java 的话（或者如果 Scala 2.12 要求 Java 1.8），选择 Java 1.8；如果使用的是 Scala 2.11，那么选择 Java 1.6 或 Java 1.7。启用自动导入功能。单击确定按钮。

Eclipse

如果使用 Eclipse 的话，那么推荐读者下载 Scala-Ide。Scala-Ide 中包含了使用 Java 或 Scala 创建本书中 Sbt/Akka 项目所需的所有插件。即使只使用 Java，读者也可能会在学习本书的过程中使用一些 Scala 相关的内容。

安装 Eclipse（Scala-Ide）

Scala-Ide 是一个在 Eclipse 中集成了 Sbt 和 Scala 插件的安装包。可以从 http://scala- ide.org 下载。

解压缩下载的文件。可以把解压缩后的文件夹移动到其他位置，比如 Linux 下的/opt 和 OSX 下的~/Applications。

运行 Eclipse 二进制可执行文件。选择一个工作目录，或是设置默认工作目录。

进入 Preferences: Java | Compiler，确认选择了正确的 Java JDK。

准备 Eclipse 项目

要在 Eclipse 中打开项目，就必须要先生成一个 Eclipse 项目。

首先，我们必须要将 eclipse sbt 插件添加到环境中。打开全局 sbt 插件文件（如果没有的话新建一个），该文件位于~/.sbt/{version}/plugins/plugins.sbt。其中 version 就是 sbt 的版本。在撰写本书时，sbt 的版本是 0.13，所以该文件为~/.sbt/0.13/ plugins/plugins.sbt。

向该文件中添加如下行，如果文件中包含多行的话，确保行与行之间以空行相隔。

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "3.0.0")
```

可能需要查看 sbteclipse 的 Git 项目，确保该插件仍然可用：https://github. com/typesafehub/sbteclipse/。

一旦安装完该插件后，就可以生成 eclipse 项目：https://github.com/ typesafehub/sbteclipse/。

在终端窗口中，进入之前建立的项目（akkademy-db）。在项目根目录下运行 activator eclipsify，生成 eclipse 项目结构。

运行成功的话，将会看到如下信息：

```
[info] Successfully created Eclipse project files for project(s): [info]
akkademy-db
```

向 Eclipse 导入项目

在 Eclipse 中，选择 File | Import。

选择 General | Existing Projects into Workspace，如图 1-6 所示。

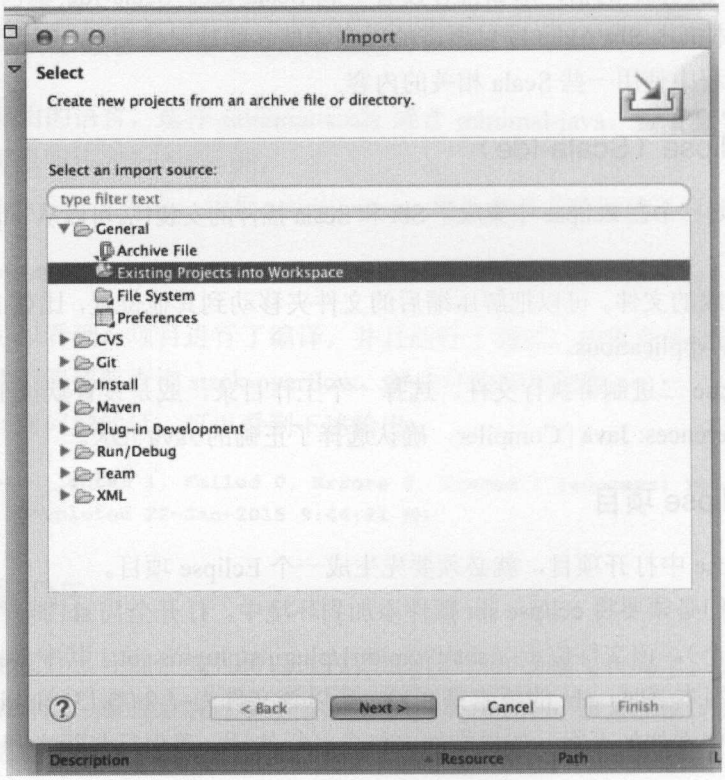


图 1-6

选择文件夹，点击下一步，如图 1-7 所示。

提示

要注意的是，如果修改了 build.sbt，那么需要重新生成项目，并且可能需要重新导入。

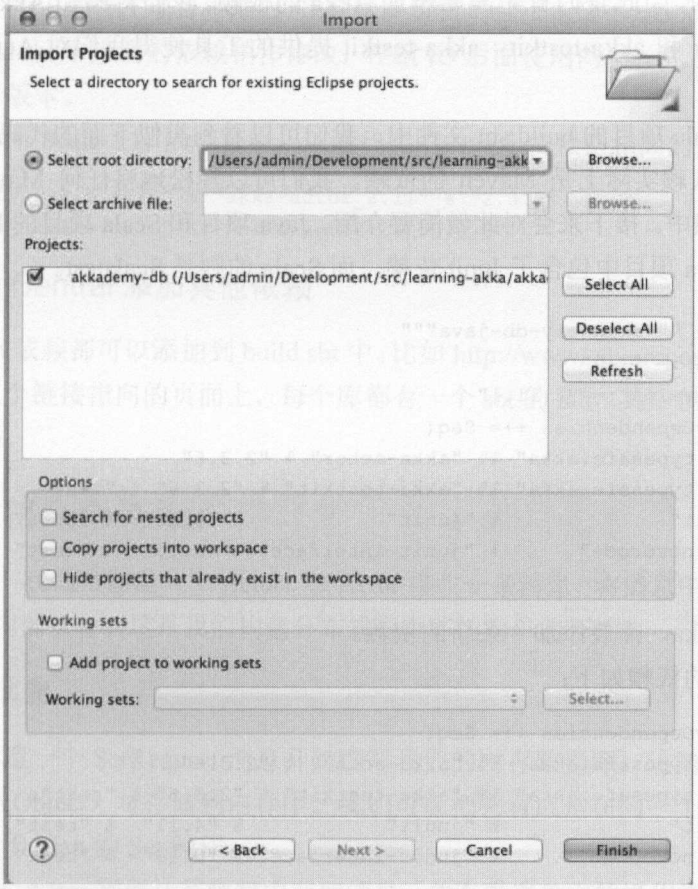


图 1-7

1.5 创建第一个 Akka 应用程序——设置 SBT 项目

既然我们已经介绍了如何配置环境和创建项目，现在就可以开始使用 Akka 来编写一些 Actor 的代码并进行测试了。我们将使用简单构建工具（Simple Build Tool，SBT）。SBT 是 Scala 项目的首选构建工具，也是 Play 框架和 Activator 实际使用的构建工具。SBT

并不复杂，而且我们只使用它来管理依赖，构建、测试并运行应用程序，因此它不会成为我们学习 Akka 的障碍。

1.5.1　将 Akka 添加至 build.sbt

现在我们将使用 IDE 打开 Java 或 Scala 应用程序。Activator 创建的项目结构并不是用于 Akka 的，因此我们首先需要添加 Akka 的依赖。我们将添加 Akka 的核心 Actor 模块 akka-actor 和 akka-testkit，akka-testkit 提供的工具使得我们对 Actor 的测试变得更加简单。

在一个 Scala 项目的 build.sbt 文件中，我们可以看到类似下面的代码。要注意的是，build.sbt 中的依赖实际上是 Maven 的依赖。我们可以轻松地将任何 Maven 依赖添加到 build.sbt 的依赖中。接下来会对此做简要介绍。Java 项目和 Scala 项目的 build.sbt 大同小异，只不过 Java 项目中包含了 Junit 依赖，而 Scala 的则是 Scalatest。

```
name := """akkademy-db-java"""
version := "1.0"
scalaVersion := "2.11.1"
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor" % "2.3.6",
  "com.typesafe.akka" %% "akka-testkit" % "2.3.6" % "test",
  "junit"             % "junit"           % "4.11"  % "test",
  "com.novocode"      % "junit-interface" % "0.10"  % "test"
  )
```

要使用 Akka，需要添加一条新的依赖。

Java 项目的依赖如下：[①]

```
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor" % "2.3.6",
  "com.typesafe.akka" %% "akka-testkit" % "2.3.6" % "test",
  "junit"             % "junit"           % "4.11"  % "test",
  "com.novocode"      % "junit-interface" % "0.10"  % "test"
)
```

Scala 项目的 build.sbt 如下：

```
name := """akkademy-db-scala"""
version := "1.0"
scalaVersion := "2.11.1"
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor" % "2.3.3",
```

① 译者注：此处原书代码有误，已更正。

```
    "com.typesafe.akka" %% "akka-testkit" % "2.3.6" % "test",
    "org.scalatest" %% "scalatest" % "2.1.6" % "test"
  )
```

使用%%获取正确的 Scala 版本

由于 Scala 的各个主要版本之间并不互相兼容，因此经常会使用多个 Scala 的版本来构建并发布库。为了能够在项目中通过 SBT 试图解析出使用正确 Scala 版本的依赖项，我们可以对 build.sbt 中声明的依赖稍作修改，在组 ID 后面使用两个%符号，而不是在库 ID 中给出 Scala 版本。

例如，在一个 Scala 2.11 的项目中，下面代码中的两个依赖是相同的：

```
"com.typesafe.akka" % "akka-actor_2.11" % "2.3.3"
"com.typesafe.akka" %% "akka-actor" % "2.3.3"
```

从 Maven Central 添加其他依赖

任何 Maven 依赖都可以添加到 build.sbt 中。比如 http://www. mvnrepository.com 中的所有依赖。在这个链接指向的页面上，每个库都有一个 sbt 的 tab，其中包含了添加 SBT 依赖需要的代码。

1.5.2 创建第一个 Actor

在本小节中，我们将创建一个 Actor，该 Actor 接收一条消息，将消息中的值存入 Map 以修改 Actor 的内部状态。这是我们构建分布式数据库的简单的开始。

首先构造消息

我们将从构造一个 SetRequest 消息开始编写我们的内存数据库。该消息用于将一个键（String）值（Object）对存储到内存中。我们可以把它看成是在同一个操作中进行插入和更新，类似于 Map 的 Set 操作。

要记住的是，Actor 需要从其邮箱中获取消息，并查看消息中的操作指示。我们通过消息的类/类型来决定具体的操作。消息类型的内容具体描述了如何实现 API 协议。在本例中，我们在消息中使用 String 作为键，Object 作为值。

消息必须永远是不可变的，这样可以确保我们和我们的团队不通过多个执行上下文/线程来做一些不安全的操作，从而避免一些奇怪而又出人意料的行为。同样要记住这些消息除了会发送给本地的 Actor 以外，也可能会发送给另一台机器上的 Actor。如果可能的话，把所有值都定义为 val（Scala）或 final（Java），并且使用不可变集合及类型，比如 Google Guava（Java）和 Scala 标准库所提供的集合及类型。

Java

下面使用 Java 将 SetRequest 消息实现为一个不可变对象。这是在 Java 中实现不可变对象的一种相当标准的方法。任何熟练的 Java 开发者应该都对此很熟悉。一般来说，我们应该始终在所有代码中优先使用不可变对象。

```
package com.akkademy.messages;

public class SetRequest {
    private final String key;
    private final Object value;

    public SetRequest(String key, Object value) {
        this.key = key;
        this.value = value;
    }

    public String getKey() {
        return key;
    }

    public Object getValue() {
        return value;
    }
}
```

Scala

在 Scala 中，我们有一种更为简洁的方式来定义不可变消息：case class。我们可以通过 case class 来创建不可变消息，一旦在构造函数中设置属性初值后，之后就只能够读取属性值，不能再进行修改：

```
package com.akkademy.messages

case class SetRequest(key: String, value: Object)
```

这样消息就定义完成了。

定义 Actor 收到消息后的响应

既然已经定义好了消息，现在我们就可以创建 Actor，并描述 Actor 接收到消息后如何做出响应。作为例子的开始，在本小节中我们将在响应中做两件事：

1. 将消息输出到日志。
2. 将任何 Set 消息中的内容保存起来，以供之后使用。

在后面的章节中，我们将在这个例子的基础上不断完善，支持获取存储的消息，并将这个 Actor 作为一个线程安全的缓存抽象层来使用（最终实现成一个全功能的分布式

键值存储)。

首先让我们看一下使用 Java 8 实现的 Actor。

Java-AkkademyDb.java

下面的代码展示了使用 Java 实现的 Actor 收到消息后的响应[①]

```
package com.akkademy;

import akka.actor.AbstractActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import akka.japi.pf.ReceiveBuilder;
import com.akkademy.messages.SetRequest;

import java.util.HashMap;
import java.util.Map;

public class AkkademyDb extends AbstractActor {
    protected final LoggingAdapter log =
        Logging.getLogger(context().system(), this);
    protected final Map<String, Object> map = new HashMap<>();

    private AkkademyDb() {
        receive(ReceiveBuilder
            .match(SetRequest.class, message -> {
                log.info("Received Set request: {}", message);
                map.put(message.getKey(), message.getValue());
            })
            .matchAny(o -> log.info("received unknown message: {}", o))
            .build()
        );
    }
}
```

Actor 是一个继承了 AbstractActor（Java8 的 Java Akka API）的 Java 类。我们在这个类里创建了 log 和 map 两个变量，并且声明为 protected，这样就可以在本章后面的测试用例中访问这两个变量。

我们在构造函数中调用 receive。receive 方法接受 ReceiveBuilder 作为参数，我们连续调用 ReceiveBuilder 的几个方法，生成最终的 ReceiveBuilder。这样就描述了 Actor 接收到不同类型消息时该如何做出响应。在这里我们定义两种响应，并逐一介绍。

① 译者注：原书代码有误，已更正，请参照 GitHub 上的代码：https://github.com/ jasongoodwin/learning-akka/blob/master/ch1/akkademy-db-java/src/main/java/com/akkademy/AkkademyDb.java。

首先，我们定义 Actor 接收到任何 SetRequest 消息时做出的响应：

```
match(SetRequest.class, message -> {
    log.info("Received Set request: {}", message);
    map.put(message.getKey(), message.getValue());
}).
```

在 Java8 的 API 里，ReceiveBuilder 的 match 方法有点儿像 case 语句，只不过 match 方法可以匹配类的类型。更正式地来说，这其实就是模式匹配。

在调用的 match 方法中，我们定义：如果消息的类型是 SetRequest.class，那么接受该消息，打印日志，并且将该 Set 消息的键和值作为一条新纪录插入到 map 中。

其次，我们捕捉其他所有未知类型的消息，直接输出到日志。

```
matchAny(o -> log.info("received unknown message"))
```

Scala-AkkademyDb.scala

由于 Scala 在语言层面原生支持模式匹配，因此用来实现 Actor 最合适不过了。现在我们来看一下相应的 Scala 代码。

```
package com.akkademy

import akka.actor.Actor
import akka.event.Logging
import scala.collection.mutable.HashMap
import com.akkademy.messages.SetRequest

class AkkademyDb extends Actor {
  val map = new HashMap[String, Object]
  val log = Logging(context.system, this)

  override def receive = {
    case SetRequest(key, value) => {
      log.info("received SetRequest - key: {} value: {}", key, value)
      map.put(key, value)
    }
    case o => log.info("received unknown message: {}", o);
  }
}
```

在 Scala API 中，我们混入了 Actor Trait，和在 Java 中一样定义了 map 和 log，然后实现 receive 方法。Actor Trait 中的 receive 方法返回一个 Receive，在 Akka 的源代码中，将 Receive 定义为一个 PartialFunction，如下所示：

```
type Receive = scala.PartialFunction[scala.Any, scala.Unit]
```

我们使用模式匹配生成的 PartialFunction 来定义接收到 SetRequest 消息时的响应。通过模式匹配提供的语义，我们可以用更清晰的代码提取出表示键和值的变量：

```
case SetRequest(key, value)
```

这个响应行为就是直接将该请求输出到日志，然后在 map 中插入键值对。

```
case SetRequest(key, value) => {
  log.info("received SetRequest - key: {} value: {}", key, value)
  map.put(key, value)
}
```

最后，我们添加一种情况用于捕捉其他所有未知类型的消息，并且直接将这样的消息输出到日志。

```
case o => log.info("received unknown message: {}", o);
```

这样就定义好 Actor 了。接下来我们需要验证编写的 Actor 是否正确。

1.5.3 使用单元测试验证代码

尽管介绍测试框架的书可能会直接打印到控制台，或是创建网页来证明代码可以正确运行，不过我们将使用单元测试来验证代码并阐释 Actor 的用法。库的代码和服务很多情况下都没有提供易于交互或理解的 API，而一般来说，几乎所有项目都通过测试来对这些库和服务进行验证。对于任何严肃的开发者来说，这都是必须具备的重要技能。

Akka Testkit

Akka 提供了一套测试工具集，其中几乎包含了测试 Actor 代码需要的所有工具。之前配置项目的时候，我们已经导入了这个测试工具集的依赖。在 build.sbt 中，它的 SBT 依赖如下所示，读者可作参考：

```
"com.typesafe.akka" %% "akka-testkit" % "2.3.6" % "test"
```

我们在这里将会使用 testkit 中提供的 TestActorRef，而不是普通的 ActorRef（下一章中将作介绍）。TestActorRef 是通用的，有两个功能：首先，它提供的 Actor API 是同步的，这样我们就不需要在测试中考虑并发的问题；其次，我们可以通过 TestActorRef 访问其背后的 Actor 对象。

说得更清楚一点，Akka 隐藏了真实的 Actor（AkkademyDb），提供了指向 Actor 的

引用，我们可以将消息发送至该引用。这样就对 Actor 进行了封装，保证了没有任何人可以直接访问真正的对象实例，只能进行消息传输。

接下来让我们来看看源代码，然后逐行解释。

Java

下面是使用 Akka testkit[①]的源代码：

```
package com.akkademy;

import static org.junit.Assert.assertEquals;

import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;
import akka.testkit.TestActorRef;
import com.akkademy.messages.SetRequest;
import org.junit.Test;

public class AkkademyDbTest {

    ActorSystem system = ActorSystem.create();

    @Test
    public void itShouldPlaceKeyValueFromSetMessageIntoMap() {
        TestActorRef<AkkademyDb> actorRef =
            TestActorRef.create(system, Props.create(AkkademyDb.class));
        actorRef.tell(
            new SetRequest("key", "value"), ActorRef.noSender());

        AkkademyDb akkademyDb = actorRef.underlyingActor();
        assertEquals(akkademyDb.map.get("key"), "value");
    }

}
```

Scala

下面是与 Actor 进行交互的源代码：

```
package com.akkademy

import akka.actor.ActorSystem
import akka.testkit.TestActorRef
import akka.util.Timeout
import com.akkademy.messages.SetRequest
import org.scalatest.{FunSpecLike, Matchers}
```

① 译者注：原文为 toolkit，有误。

```
import scala.concurrent.duration._

class AkkademyDbSpec extends FunSpecLike with Matchers {
  implicit val system = ActorSystem()
  implicit val timeout = Timeout(5 seconds)

  describe("akkademyDb") {
    describe("given SetRequest") {
      it("should place key/value into map") {
        val actorRef = TestActorRef(new AkkademyDb)
        actorRef ! SetRequest("key", "value")

        val akkademyDb = actorRef.underlyingActor
        akkademyDb.map.get("key") should equal(Some("value"))
      }
    }
  }
}
```

这是我们第一次讲到与 Actor 的交互，所以有一些之前没出现过的代码。其中一部分是专门用于测试的，另一部分是与 Actor 的交互相关的。

我们已经介绍过，Actor 系统是包含了所有 Actor 及其地址的一个地方，所以在创建 Actor 之前，需要做的第一件事就是获取 Actor 系统的引用。在上面的测试示例里，我们将该引用存在一个变量中：

```
//Java
ActorSystem system = ActorSystem.create();
//Scala
implicit val system = ActorSystem()
```

在创建完 Actor 系统之后，我们就可以在 Actor 系统中创建 Actor 了。正如之前提到的那样，我们将使用 Akka Testkit 来创建一个 TestActorRef，提供同步 API，并且允许我们访问其指向的 Actor。下面，我们就在 Actor 系统中创建 Actor：

```
//Java
TestActorRef<AkkademyDb> actorRef = TestActorRef.create(system, Props.
create(AkkademyDb.class));
//Scala
val actorRef = TestActorRef(new AkkademyDb)
```

我们调用 Akka Testkit 中 TestActorRef 提供的 create 方法，传入创建的 Actor 系统（在 Scala 中是隐式传入的），以及指向 Actor 类的引用。在后面的章节中，我们将介绍如何创建 Actor。由于真正的 Actor 实例被隐藏了，因此在我们的 Actor 系统中创建 Actor 返

回的是一个 ActorRef（在本例中是 TestActorRef），我们可以将消息发送至该 ActorRef。有了 Actor 系统和 Actor 的类引用后，Akka 就足以在 Actor 系统的创建这个简单的 Actor。因此，现在我们已经成功地创建了第一个 Actor 了。

与 Actor 之间的交互是通过消息传递来进行的。我们使用“tell”或是 Scala 中的“!”（仍然读作“tell”）将消息放入 Actor 的邮箱中。使用 Java 时，我们通过 tell 的第二个参数定义该消息并不需要任何响应对象。而在 Scala 中，这是在外部隐式定义的。

```
//Java
actorRef.tell(new SetRequest("key", "value"), ActorRef.noSender());
//Scala
actorRef ! SetRequest("key", "value")
```

因为我们使用的是 TestActorRef，所以只有在 tell 调用请求处理完成后，才会继续执行后面的代码。对于我们的第一个 Actor 来说，这并没有问题，但是要注意的是，这个例子并没有展示出 Actor API 的异步特性，而这并不是常见的用法。通常情况下，tell 是一个异步操作，调用后会立即返回。

最后，我们需要检查 Actor 是否将值存入了 map 中，确认其行为是否正确。为了进行这项检查，首先得到指向背后 Actor 实例的引用，然后调用 get("key")检查 map，确认已经将值存入 map 中。

```
//Java
AkkademyDb akkademyDb = actorRef.underlyingActor();
assertEquals(akkademyDb.map.get("key"), "value");
//Scala
val akkademyDb = actorRef.underlyingActor
akkademyDb.map.get("key") should equal(Some("value"))
```

这样我们就完成了第一个简单的测试用例。这个基本的模式可以用于构建同步的 Actor 单元测试。在阅读本书的过程中，我们会看到更多的单元测试示例以及对 Actor 进行异步集成测试的例子。

1.5.4　运行测试用例

我们快要完成了！既然已经创建了测试用例，我们就可以打开命令行，运行“activator”，启动 activator cli。接着运行“clean”，删除所有无用文件，然后运行“test”，开始执行测试用例。可以通过运行 activator clean test 在一条命令中执行上述所有操作。

运行 Java 的 JUnit 测试用例时，会看到如下输出：

```
[INFO] [01/12/2015 23:09:24.893] [pool-7-thread-1] [akka://default/
user/$$a]
Received Set request: Set{key='key', value=value}
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1
[success] Total time: 7 s, completed 12-Jan-2015 11:09:25 PM
```

而如果使用 Scala 的话，scala-test 的输出会更优雅一些：

```
[info] AkkademyDbSpec:
[info] akkademyDb
[info] - should place key/value from Set message into map
[info] Run completed in 1 second, 990 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
```

从输出结果中，我们可以看到运行测试用例的总数和失败的测试用例数。如果有错误的话，结果中会给出错误发生的位置，以便我们进行排查。一旦编写了测试用例并指定了其应有的行为，就不用担心对代码的修改或重构会导致错误的行为了。

1.6 课后作业

为了确保读者能够很好地掌握每章的内容，在每章的结尾将会给出一些课后作业。

- 将 Akka 文档添加至浏览器的书签。把 Hacker News 也添加至书签，每天都阅读一下。
- 设想一个想要编写并发布到互联网上的服务。最好是需要对一些输入进行处理、保存并返回响应的服务。
- 到 GitHub 上为要做的项目创建一个代码库，并将项目与 GitHub 保持同步。如果读者从来没有使用过 Git 或是 GitHub 的话，由于本书的所有源代码都在 GitHub 上，因此这是个不错的学习机会。读者可以使用 GitHub 来发布并显示在本书中完成的工作。给项目的 README 加上 LEARNINGAKKAJG 的标签，这样就能让别人在 GitHub 上搜索到我们所完成的工作。
- 创建一个 Actor，并使其保存最近发送出去的字符串。
- 编写一个测试用例，确认 Actor 能够正确地接收消息。
- 编写一个测试用例，确认 Actor 在接收到两个消息时的行为仍然正确。

- 将项目推送到 GitHub。
- 从 http://www.github.com/jasongoodwin/learning-akka 下载本书的源代码。

1.7　小结

我们已经正式开始学习如何 Akka 来编写可扩展的分布式应用程序了。本章简要介绍了 Actor 模型的历史，以帮助我们了解 Akka 的含义和由来。接着，讲述了如何创建用于 Akka 代码的 SBT 项目。我们配置了开发 SBT 项目所需的环境，并创建了一个 Actor。然后通过一个测试用例对创建的 Actor 进行了测试。

在接下来的章节中，我们会给示例应用程序添加一个客户端，并将其分布到多个核以及处理器上，这个程序会开始看上去像一个真正的分布式数据库。

第2章
Actor与并发

本章将介绍进行并发编程及异步编程所需的背景知识。在继续学习后面的章节前，理解本章的内容至关重要。如果读者早已熟稔Scala的Future，Play的Promise或是Java 8的CompletableFuture，那么可以跳过本章。如果曾经使用过Guava或Spring的Listenable Future，可能需要了解本章所介绍API的不同之处。如果从来没有使用过monadic风格的Future，那就需要花点时间学习一下本章了。

本章涉及如下话题：

- 对Actor的剖析、Actor的创建以及与Actor的通信；
- 使用Akka API处理异步响应所需的工具和知识；
- Future（用于存放将来可能成功或失败的结果）的使用。

2.1 响应式系统设计

任何关于Akka的书籍都会有关于“响应式”这个术语的描述。Akka也被称作是一个响应式平台，更具体的说，它是Typesafe响应式平台的一部分。由此而论，这个术语变得越来越流行，尤其要感谢响应式宣言（Reactive Manifesto）。响应式宣言是一份文档，尝试分析Web应用程序要满足如今用户的需要以成功地进行扩展所必需的特性。

从这个角度来看，这个词其实是开发者的一个流行文化。当然，也有一些人在听到这个词的时候可能会对它颇有微词。

本小节将简要介绍响应式宣言中提出的4个准则。这些是我们在开发应用程序的过程中需要努力支持的特性，以提高程序的可扩展性和容错性。在介绍后面的内容时，我们也会回过头来提到这些特性。可以从http://reactivemanifesto.org/访问查看响应式宣言。

2.2　响应式四准则

响应式宣言中包含了 4 个准则，或者说是设计目标：灵敏性、伸缩性、容错性以及事件驱动设计。接下来通过应用实例分别对这 4 个准则进行概述。

2.2.1　灵敏性

应用程序应该尽可能快地对请求做出响应。

如果可以在顺序获取数据和并行获取数据之间进行选择的话，为了尽快向用户返回响应，始终应该优先选择并行获取数据，可以同时请求互相没有关联的数据。当我们需要请求多个互相无关、没有依赖的数据时，应该考虑是否能够同时请求这些数据。

如果可能出现错误，应该立即返回，将问题通知用户，不要让用户等待直到超时。

2.2.2　伸缩性

应用程序应该能够根据不同的工作负载进行伸缩扩展（尤其是通过增加计算资源来进行扩展）。为了提供伸缩性，系统应该努力消除瓶颈。

如果在虚拟机上运行内存数据库，那么添加另一个虚拟节点就可以将所有的查询请求分布到两台虚拟服务器上，将可能的吞吐量增加至原来的两倍。添加额外的节点应该能够几乎线性地提高系统的性能。

增加一个内存数据库的节点后，还可以将数据分为两半，并将其中的一半移至新的节点，这样就能够将内存容量提高至原来的两倍。添加节点应该能够几乎线性地提高内存容量。

2.2.3　容错性

应用程序应该考虑到错误发生的情况，并且从容地对错误情况做出响应。如果系统的某个组件发生错误，对与该组件无关的请求不应该产生任何影响。错误是难以避免的，因此应该将错误造成的影响限制在发生错误的组件之内。如果可能的话，通过对重要组件及数据的备份和冗余，这些组件发生错误时不应该对其外部行为有任何影响。

2.2.4　事件驱动/消息驱动

使用消息而不直接进行方法调用提供了一种帮助我们满足另外 3 个响应式准则的方法。消息驱动的系统着重于控制何时、何地以及如何对请求做出响应，允许做出响应的组件进行路由以及负载均衡。

由于异步的消息驱动系统只在真正需要时才会消耗资源（比如线程），因此它对系统资源的利用更为高效。消息也可以被发送到远程机器（位置透明）。因为要发送的消息暂存在 Actor 外的队列中，并从该队列中发出，所以就能够通过监督机制使得发生错误的系统进行自我恢复。

2.2.5 响应式准则的相关性

4 个响应式准则之间并不是完全独立的。为了满足某个准则而采取的方法通常也对满足其他准则有所帮助。例如，如果发现某个服务响应速度较慢，我们可能会在短时间内停止再向该服务发送请求，等待其恢复正常，并立即向用户返回错误信息。这样做降低了响应慢的服务不堪重负直接崩溃的风险，因此也提高了系统的容错性。除此之外，我们立即告知了用户系统发生的问题，也就改善了系统的响应速度，如图 2-1 所示。

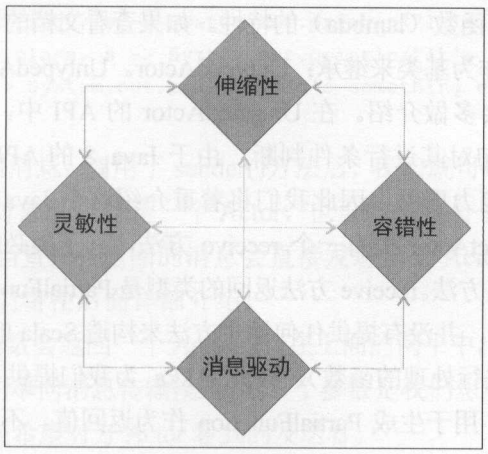


图 2-1

2.3 剖析 Actor

在介绍与本书要编写的分布式数据库更相关的例子之前，我们将先介绍 Actor 最基本的特性，理解 Actor 的基本结构和方法。为了展示最简单的可能情况，我们将在这个例子中构建一个简单的 Actor，这个 Actor 接收一个字符串“Ping”，返回字符串“Pong”作为响应。

2.3.1 Java Actor API

我们将首先介绍 Java 实现。Java 和 Scala 的 API 差别很大，因此需要分别介绍。

```
public class JavaPongActor extends AbstractActor {
    public PartialFunction receive() {
        return ReceiveBuilder
            .matchEquals("Ping", s ->
                sender().tell("Pong", ActorRef.noSender()))
            .matchAny(x ->
                sender().tell(
                    new Status.Failure(new Exception("unknown message")), self()
            ))
            .build();
    }
}
```

上面的代码展示了 Java 的 Actor API。我们将具体解释这个例子。

- AbstractActor：首先，我们继承了 AbstractActor。这是一个 Java 8 特有的 API，利用了 Java 8 的匿名函数（lambda）的特性。如果查看文档的话，可以发现还有另一个 Actor API 可以作为基类来继承：UntypedActor。UntypedActor 是较老版本的 API，在本书中并不会多做介绍。在 UntypedActor 的 API 中，会得到一个对象，然后必须使用 if 语句对其进行条件判断。由于 Java 8 的 API 通过匿名函数来实现模式匹配，表达能力更强，因此我们将着重介绍这个 Java 8 特有的 API。
- Receive：AbstractActor 类有一个 receive 方法，其子类必须实现这个方法或是通过构造函数调用该方法。receive 方法返回的类型是 PartialFunction，这个类型来自 Scala API。在 Java 中，并没有提供任何原生方法来构造 Scala 的 PartialFunction（并不对所有可能输入进行处理的函数），因此 Akka 为我们提供了一个抽象的构造方法类 ReceiveBuilder，用于生成 PartialFunction 作为返回值。不用担心，使用 Akka 并不要求理解 Scala 的 PartialFunction！
- ReceiveBuilder：连续调用 ReceiveBuilder 的方法，为所有需要匹配处理的输入消息类型提供响应方法的描述，然后调用 build() 方法生成需要返回的 PartialFunction。
- Match：ReceiveBuilder 提供了一些值得一提的 match 方法，我们将提供一些示例，展示如何分别使用这些方法来匹配“ping”消息。
  - match(class, function)：描述了对于任何尚未匹配的该类型的示例，应该如何响应。

```
match(String.class, s -> {if(s.equals("Ping")) respondToPing(s);})
```

  - match(class, predicate, function)：描述了对于 predicate 条件函数为真的某特

定类型的消息，应该如何响应。

```
match(String.class, s -> s.equals("Ping"), s -> respondToPing(s))
```

- matchEquals(object, function)：描述了对于和传入的第一个参数相等的消息，应该如何响应。

```
matchEquals("Ping", s -> respondToPing(s))
```

- matchAny(function)：该函数匹配所有尚未匹配的消息。通常来说，最佳实践是返回错误信息，或者至少将错误信息记录到日志，帮助开发过程中的错误调试。

match 函数从上至下进行模式匹配。所以可以先定义特殊情况，最后定义一般情况。

```
ReceiveBuilder
    .matchEquals("Ping", s -> System.out.println("It's Ping: " + s))
    .match(String.class, s -> System.out.println("It's a string: " + s))
    .matchAny(x -> System.out.println("It's something else: " + x))
    .build
```

- 向 sender()返回消息：调用了 sender()方法后，我们就可以返回所收到的消息的响应了。响应的对象既可能是一个 Actor，也可能是来自于 Actor 系统外部的请求。第一种情况相当直接：返回的消息会直接发送到该 Actor 的收件信箱中。对于第二种情况，我们将在后面详细介绍。
- tell()：sender()函数会返回一个 ActorRef。在上面的例子中，我们调用了 sender().tell()。tell()是最基本的单向消息传输模式。第一个参数是我们想要发送至对方信箱的消息。第二个参数则是希望对方 Actor 看到的发送者。

  和第 1 章初识 Actor 中使用过的方法类似，我们描述了接收到的消息是 String 时应该做出的响应。由于需要检查接收到的字符串是否为“Ping”，因此需要进行判断，所以这里使用的 match 方法略有不同。然后描述响应行为：通过 tell()方法向 sender()返回一条消息。我们返回的消息是字符串“Pong”。Java 的 tell 方法要求提供消息发送者的身份：这里使用 ActorRef.noSender()表示没有返回地址。
- 返回 akka.actor.Status.Failure：为了向发送方报告错误信息，需要向其发送一条消息。如果 Actor 中抛出了异常，就会通知对其进行监督的 Actor（将在第 3 章传递消息中进行介绍）。不过无论如何，如果想要报告错误消息，需要将错误发送给发送方。如果发送方使用 Future 来接收响应，那么返回错误消息会导致 Future 的结果为失败。我们将之后对此进行简要介绍。

上面几点涉及了 Java AbstractActor 的基本 API。接下来我们将了解一下 Scala API。

2.3.2　Scala Actor API

下面的代码展示了一个 Scala Actor：

```
//Scala
class ScalaPongActor extends Actor {
  override def receive: Receive = {
    case "Ping" => sender() ! "Pong"
    case _ =>
      sender() ! Status.Failure(new Exception("unknown message"))
  }
}
```

接下来我们就详细介绍这个 Actor。

- Actor：要定义一个 Actor，首先要继承 Actor 基类。Actor 基类是基本的 Scala Actor API，非常简单，并且符合 Scala 语言的特性。
- Receive：在 Actor 中重写基类的 receive 方法。并且返回一个 PartialFunction。要注意的是，receive 方法的返回类型是 Receive。Receive 只不过是定义的一种类型，表示 scala.PartialFunction[scala.Any, scala.Unit]。如果读者不是非常熟悉 Scala API 中的 PartialFunction，也不必担心，只需要构造一些模式匹配的 case 语句，每个语句都返回 Unit，并且知道这些语句并不一定要覆盖所有的可能情况即可。如果想要知道 receive 方法背后的实现原理，可以在 REPL 中做一些 scala.PartialFunction 的实验。

```
scala> val pf: PartialFunction[Any, Unit] = {
          case _: String => println("Got a String")
          }
       pf: PartialFunction[Any,Unit] = <function1>

scala> pf("hey")
Got a String
```

- 向 sender()返回消息：在示例 Actor 代码中，我们接着通过 sender()方法获取了发送者的 ActorRef。我们可以向该 ActorRef 发送消息，对发送者做出响应。在这个例子中，返回了“Pong”。
- tell 方法（!）：我们使用 tell 方法向发送方发送响应消息。在 Scala 中，通过“!”来调用 tell 方法。如果读者看了 Java 的部分，那么会注意到在 Java API 的 tell 方法中必须指定消息的发送者，不过在 Scala 中，消息发送者是隐式传入的，因

此我们不需要再显式传入消息发送者的引用。在 tell 方法“!”的方法签名中，有一个隐式的 ActorRef 参数。如果在 Actor 外部调用 tell 方法的话，该参数的默认值会设为 noSender。下面就是该方法的签名：

```
def !(message: Any)(implicit sender: ActorRef = Actor.noSender): Unit
```

- Actor 中有一个隐式的变量 self，Actor 通过 self 得到消息发送者的值。因此 Actor 中 tell 方法的消息发送者永远是 self。

```
implicit final val self = context.self
```

消息发送者是隐式传入的，无需担心，但是在对 Java 和 Scala 的 API 进行比较时，理解隐式的值到底从何而来对我们是很有帮助的。

- **返回 akka.actor.Status.Failure**：最后需要注意的是，在收到未知消息时，返回 akka.actor.Status.Failure。Actor 本身在任何情况下都不会自己返回 Failure（即使 Actor 本身出现错误）。因此如果想要将发生的错误通知消息发送者，那么我们必须要主动发送一个 Failure 给对方。发送回 Failure 会导致请求方的 Future 被标记为失败。

我们很快就会介绍如何获取另一个 Actor 发回的响应。在这之前，我们首先要能创建 Actor。

2.4 Actor 的创建

访问 Actor 的方法和访问普通对象的方法有所不同。我们从来都不会得到 Actor 的实例，从不调用 Actor 的方法，也不直接改变 Actor 的状态，反之，只向 Actor 发送消息。除此之外，我们也不会直接访问 Actor 的成员，而是通过消息传递来请求获取关于 Actor 状态的信息。使用消息传递代替直接方法调用可以加强封装性。Alan Kay 是最早对面向对象编程进行描述的人，他实际上把消息传递也定义为面向对象编程的一部分。我想 Alan Kay 看到面向对象后来的发展情况，有时候一定会唏嘘不已。

我发明了“面向对象”这个术语，不过我可以告诉你，这跟 C++一点关系都没有。

——Alan Kay, OOPSLA '97

通过使用基于消息的方法，我们可以相当完整地将 Actor 的实例封装起来。如果只通过消息进行相互通信的话，那么永远都不会需要获取 Actor 的实例。我们只需要一种机制来支持向 Actor 发送消息并接收响应。

在 Akka 中，这个指向 Actor 实例的引用叫做 ActorRef。ActorRef 是一个无类型的引用，将其指向的 Actor 封装起来，提供了更高层的抽象，并且给用户提供了一种与 Actor 进行通信的机制。

上文已经介绍过，Actor 系统就是包含所有 Actor 的地方。有一点可能相当明显：我们也正是在 Actor 系统中创建新的 Actor 并获取指向 Actor 的引用。actorOf 方法会生成一个新的 Actor，并返回指向该 Actor 的引用。

```
//Java
ActorRef actor = actorSystem.actorOf(Props.create(JavaPongActor.class));
//Scala
val actor: ActorRef =
actorSystem.actorOf(Props(classOf[ScalaPongActor]))
```

注意

要注意的是，这里我们实际上并没有新建 Actor，例如，我们没有调用 actorOf(new PongActor)。

Actor 都被封装起来了，不能够被直接访问。我们不能从外部代码中直接访问 Actor 的状态。创建 Actor 的模式保证了这一点，现在我们就来一窥究竟。

Props

为了保证能够将 Actor 的实例封装起来，不让其被外部直接访问，我们将所有构造函数的参数传给一个 Props 的实例。Props 允许我们传入 Actor 的类型以及一个变长的参数列表。

```
//Java
Props.create(PongActor.class, arg1, arg2, argn);
//Scala
Props(classOf[PongActor], arg1, arg2, argn)
```

如果 Actor 的构造函数有参数，那么推荐的做法是通过一个工厂方法来创建 Props。假如我们不希望 Pong Actor 返回“Pong”，而是希望其返回另一条消息，那么可能就会需要这样的构造参数。我们可以创建一个工厂方法，用于生成这样的 Props 示例：

```
//Java
public static Props props(String response) {
    return Props.create(this.class, response);
```

```
}
//Scala
object ScalaPongActor {
  def props(response: String): Props = {
    Props(classOf[ScalaPongActor], response)
  }
}
```

然后就可以使用 Props 的工厂方法来创建 Actor：

```
//Java
ActorRef actor = actorSystem.actorOf(JavaPongActor.props("PongFoo"));
//Scala
val actor: ActorRef = actorSystem.actorOf(ScalaPongActor props
"PongFoo")
```

虽然创建 Props 的工厂方法并非必须，但是能够在同一个地方管理对 Props 对象的创建，因此所有对 Actor 构造参数的修改都可以与其他代码隔离，防止在代码修改的过程中引起其他模块的错误。

actorOf 创建一个 Actor，并返回指向该 Actor 的引用 ActorRef。除此之外，还有另一种方法可以获取指向 Actor 的引用：actorSelection。

为了理解 actorSelection，我们需要先来看一下 Actor 的路径。每个 Actor 在创建时都会有一个路径，我们可以通过 ActorRef.path 来查看该路径。该路径看上去如下所示：

- akka://default/user/BruceWillis

该路径是一个 URI，它甚至可以指向使用 akka.tcp 协议的远程 Actor。

- akka.tcp://my-sys@remotehost:5678/user/CharlieChaplin

要注意的是，路径的前缀说明使用的协议是 akka.tcp，并且指定了远程 Actor 系统的主机名和端口号。如果知道 Actor 的路径，就可以使用 actorSelection 来获取指向该 Actor 引用的 ActorSelection（无论该 Actor 在本地还是远程）。

```
ActorSelection selection = system.actorSelection("akka.tcp://
   actorSystem@host.jason-goodwin.com:5678/user/KeanuReeves");
```

这里的 ActorSelection 是一个指向 Actor 的引用，我们可以像使用 ActorRef 一样，使用 ActorSelection 进行网络间的通信。由此可以看出，使用 Akka 在网络上传递消息非常容易，而这也是 Akka 的位置透明性在实际应用中的一个例子。要对某个应用程序进行修改，使之与远程服务进行通信，几乎只需要修改对 Actor 位置的配置就足够了。

现在来回顾一下本小节的内容。我们可以创建一个 Actor，传入一个构造参数列表构建一个 Props 实例，并将该 Props 实例作为参数传给 system.actorOf 并调用 system.actorOf 方法，得到指向该 Actor 的引用。要为 Actor 指定名字的话，只需要将该名字作为参数传给 actorOf 方法即可。最后，我们可以使用 actorSelection 在本地或远程系统上查找已有的 Actor。

接下来，我们将开始学习如何编写异步和事件驱动的代码。

2.5　Promise、Future 和事件驱动的编程模型

在继续学习更复杂的基于 Actor 的应用程序之前，需要了解一些事件驱动编程模型中的基本抽象概念：Promise 和 Future。在第一章初识 Actor 中，我们已经了解过如何向一个 Actor 发送消息，以及如何在 Actor 内根据接收到的事件进行不同的响应行为。但是，如果想要通过发送消息向 Actor 请求获取一些输出结果呢？比如说需要从内存键值存储中获取一条记录。

2.5.1　阻塞与事件驱动 API

几乎每个开发者都很熟悉阻塞式的代码。进行 IO 操作时，编写的都是阻塞式的代码。当我们调用一个同步的 API 时，调用的方法不会立即返回：应用程序会等待该调用执行完成。例如，如果发起一个 HTTP 请求的话，只有在请求完成后，才会收到返回的响应对象。由于发起调用的线程会暂停执行并等待，因此等待 IO 操作完成的代码都是阻塞的，在 IO 操作完成之前，发起调用的线程无法进行任何其他操作。下面是一个阻塞式代码的例子。例子使用 Java 数据库连接（Java Database Connectivity，JDBC）发起一个查询：

```
stmt = conn.createStatement();
String sql = "select name from users where id='123'";
ResultSet rs = stmt.executeQuery(sql);
rs.next();
String name  = rs.getString("name");
```

这里我们使用 JDBC 从数据库获取一个用户名。代码看上去非常简单，但是有一些不是很显然的运行行为降低了这段简单代码的可读性。

- 延时：并非立即能够获得通过网络传回的结果。
- 失败：请求可能会失败（例如远程服务有可能不可用）。有许多可能性都会导致抛出异常。

调用 executeQuery 时，发起调用的线程必需等待数据库查询的完成。在一个 Web 应用程序中，有许多用户可能会同时发起很多并发请求，线程池中的线程数很可能会达到其支持的最大值。如果这些线程都在等待 IO 操作完成的话，那么即使还有可用的计算资源，也没有任何线程能够使用这些资源，因此服务器也就无法再进行任何操作了。如果读者曾经对阻塞式的基于 Servlet 的 Web 应用程序做过性能调优，那么可能接触过线程池中线程数的最大限制问题。通常情况下，在服务器负载较大时，由于所有线程都只是在等待，服务器无法充分利用 CPU 资源。

这可能是因为线程池中的线程被耗尽，也可能是因为系统把时间都花在了需要 CPU 的线程之间的上下文切换上，而没有利用 CPU 进行实际的工作。同样，由于线程池中的线程数是有限的，因此如果所有线程都在等待的话，服务器在释放其中一个线程资源之前就无法处理接收到的其他请求，导致系统响应延时增加。

因此，读者有可能会问，为什么不直接使用没有线程数量限制的线程池呢（为每一个请求都新建一个线程）？新建线程是有开销的，维护许多运行的线程也是有开销的。在同一个 CPU 核心中运行多个线程时，操作系统需要不断切换线程上下文，保证所有的线程都能分配到 CPU 时间。CPU 需要获取并存储当前线程的状态，然后载入下一个需要使用 CPU 的线程的上下文。如果运行了 1000 个线程的话，可以想象，大把的时间会耗费在上下文切换上。

总结一下，使用多线程来处理阻塞式 IO 时会遇到一些问题：

- 代码没有在返回类型中明确表示错误；
- 代码没有在返回类型中明确表示延时；
- 阻塞模型的吞吐量受到线程池大小的限制；
- 创建并使用许多线程会耗费额外的时间用于上下文切换，影响系统性能。

非阻塞、异步的消息驱动系统可以只运行少量的线程，并且不阻塞这些线程，只在需要计算资源时才使用它们。这大大提高了系统的响应速度，并且能够更高效地利用系统资源。取决于具体实现的不同，异步系统还可以在返回类型中清晰地定义错误和延时等信息，我们接下来就会看到这一点带来的好处。

其缺点在于我们需要花点时间来理解如何使用消息驱动的编程模型来编写代码。对于这两种模型，我们将各给出一个例子，帮助我们更好地理解这两种设计方法的工作原理。

首先，我们来看一个非常简单的使用阻塞 IO 调用数据库的例子。

```
//Java
String username = getUsernameFromDatabase(userId);
System.out.println(username);
```

```
//Scala
val username = getUsernameFromDatabase(userId)
println(username)
```

调用了方法之后，线程会进入到被调用的方法，得到结果后再返回。

如果进行调试的话，可以设置断点在该线程内进入被调用的 getUsernameFromDatabase 方法，逐行查看该方法的具体执行情况。一旦开始执行真正的 IO 操作，该线程就会暂停，直到 IO 结果返回为止。然后，线程返回该方法的结果，跳出该方法，继续执行，打印出结果，如图 2-2 所示。

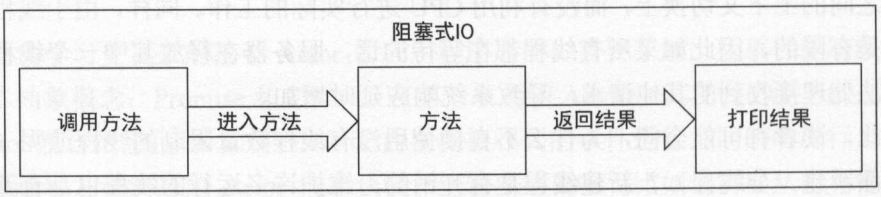


图 2-2

基于事件驱动编写相同功能的代码时，由于需要描述事件完成时要进行的操作，而该操作在一个不同的上下文中执行，因此代码看上去会有所差别。要把上面的例子改为事件驱动，就需要在数据库返回结果之后再在代码中调用打印语句。刚开始可能需要花点时间来适应这个模型，但是一旦适应以后就会觉得很自然了。

要转而使用事件驱动的模型，我们需要在代码中用不同的方法来表示结果。我们需要用一个占位符来表示最终将会返回的结果：Future。然后注册事件完成时应该进行的操作：打印结果。我们注册的代码会在 Future 占位符的值真正返回可用时被调用执行。“事件驱动”这个术语正是描述了这种方法：在发生某些特定事件时，就执行某些对应的代码。

```
//Java
CompletableFuture<String> usernameFuture =
    getUsernameFromDatabaseAsync(userId);
usernameFuture.thenRun(username ->
    //executed somewhere else
    System.out.println(username)
);
//Scala
val future = getUsernameFromDatabaseAsync(userId)
future.onComplete(username =>
  //executed somewhere else
  println(username)
)
```

从线程的角度来看，代码首先会调用方法，然后进入该方法内部，接着几乎立即返回一个 Future/CompletableFuture。返回的这个结果只是一个占位符，真正的值在未来某个时刻最终会返回到这个占位符内。

我们不会过于详细地介绍这个方法调用本身，读者需要理解的是：该方法会立即返回，而数据库调用及结果的生成是在另一个线程上执行的。ExecutionContext 表示了执行这些操作的线程，我们将在本书后面的章节中对此进行介绍。（在 Akka 中，可以看到 ActorSystem 中有一个 dispatcher，就是 ExecutionContext 的一种实现。）

要注意的是，调试异步代码与调试同步代码有着很大的不同：我们无法在发起调用的线程上看到数据库调用的所有细节，因此无法像调试使用阻塞模型的代码一样，使用调试器在发起调用的线程内单步执行，了解数据库调用的所有细节。同样地，如果查看某个错误信息的栈追踪信息，可能找不到最开始发起调用的代码，看到的是真正执行这段代码的栈信息。

方法返回 Future 之后，我们只得到了一个承诺，表示真正的值最终会返回到 Future 中。我们并不希望发起调用的线程等待返回结果，而是希望其在真正的结果返回后再执行特定的操作（打印到控制台）。在一个事件驱动的系统中，需要做的就是描述某个事件发生时需要执行的代码。在 Actor 中，描述接收到某个消息时进行的操作。同样地，在 Future 中，我们描述 Future 的值真正可用时进行的操作。在 Java 8 中，使用 thenRun 来注册事件成功完成时需要执行的代码；而在 Scala 中，使用 onComplete，如图 2-3 所示。

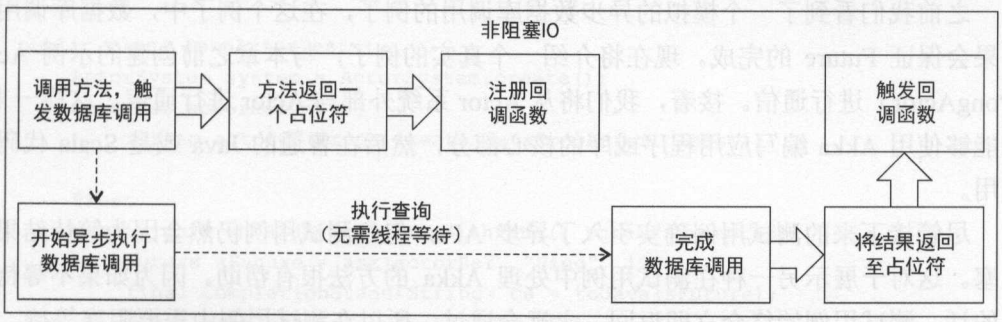


图 2-3

有一点必须要再次强调：打印语句并不会运行在进行事件注册的线程上。它会运行在另一个线程上，该线程信息由 ExecutionContext 维护。Future 永远是通过 Execution Context 来创建的，因此我们可以选择在哪里运行 Future 中真正需要执行的代码。

在注册的匿名函数中，可以访问到作用域内的所有变量。不过由于方法并不在与闭包相同的同一词法作用域内被调用，因此在调用方法时要格外小心，或者直接不要在闭包内调用方法。我们将会在下一章中介绍这一要点。

要注意的是，Future 是有可能执行失败的，因此一定要给 Future 提供一个超时参数（在 Scala API 中是必须提供的），这样一来，Future 就不可能一直等待结果，不管是执行成功，还是失败，可以保证 Future 一定会执行完成。接下来我们会更深入地介绍如何来处理 Future。

技能点检查

Scala 开发者应该对高阶函数和匿名函数的使用较为熟悉。这对于开发者能够迅速地理解本书的内容是很有帮助的。

如果读者正在使用 Java 8，那么应该下载本书目前为止的代码示例，检查自己阅读这些示例时是否遇到困难。如果对匿名函数的使用尚且不太清楚，那么读者现在就应该花点时间过一遍 Oracle 的 Java 8 匿名函数简介，可以从下面的链接处找到该教程：
http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/
Lambda-QuickStart/index.html

读者还应该了解一下 Stream API 和 Optional 类型，并做一些实践练习。由于 Optional 类型的使用在语义上和 CompletableFuture 类似，因此了解 Optional 是很有帮助的。

有了上面这些相关经验之后，再理解下面的内容就容易多了。

2.5.2　使用 Future 进行响应的 Actor

之前我们看到了一个模拟的异步数据库调用的例子，在这个例子中，数据库调用的结果会保证 Future 的完成。现在将介绍一个真实的例子，与本章之前创建的示例 Actor（PongActor）进行通信。接着，我们将从 Actor 系统外部与 Actor 进行通信，这样一来，就能够使用 Akka 编写应用程序或库的核心部分，然后在普通的 Java 或是 Scala 代码中使用。

尽管接下来的测试用例确实引入了异步 API，但是测试用例仍然会因为等待结果而阻塞。这对于展示另一种在测试用例中处理 Akka 的方法很有帮助。因为如果不等待结果的话，测试用例始终会立即返回，永远会通过，所以在测试用例中需要阻塞等待。

读者应该编写好这些测试用例，并且通过这些测试用例来更好地了解并理解下一小节中将介绍的 Future API。有了这些例子以后，就算是将来需要更深入地学习 Future 的工作原理，始终可以回过来好好利用这些测试用例。这些例子也包含在本书的源代码中。

Java 示例

我们将首先介绍使用 Java 8 的例子。Akka 是用 Scala 编写的。一般来说，Scala 和 Java 的 API 是一一对应的，不过有一个重要的特例：所有返回 Future 的异步方法返回的

都是 Scala 的 scala.concurrent.Future。

处理 Scala 的 Future

在使用 Java 的例子中，我们需要某种方法来处理 Scala 的 Future。在本书中，我们将把 Scala 的 Future 转换成 Java 8 的 CompletableFuture。

如果在构建 Play 应用程序的话，Play 的 Promise API 也是个很好的选择。相比于 Java 8 的 CompletableFuture API，笔者个人更喜欢 Play 的 Promise API 使用的语义。不过对于还不习惯编写异步代码的人来说，可能 Java 8 的 API 可读性更高一些。如果编写的代码会成为某个库的一部分，那么推荐使用 Java 8 的 CompletableFuture，这样在代码中就不会有对 Play 的外部依赖了。

首先，需要在 build.sbt 中加入一个 Scala 团队提供的依赖项，支持 Scala 和 Java 8 Future 之间的相互转换。

```
"org.scala-lang.modules" %% "scala-java8-compat" % "0.6.0"
```

测试用例

下面就是完整的测试用例。接下来我们就详细介绍 API 的各个部分。

```
package pong;
//[...imports]
import static scala.compat.java8.FutureConverters.*;

public class PongActorTest {
    ActorSystem system = ActorSystem.create();
    ActorRef actorRef =
        system.actorOf(Props.create(JavaPongActor.class));

    @Test
    public void shouldReplyToPingWithPong() throws Exception {
        Future sFuture = ask(actorRef, "Ping", 1000);
        final CompletionStage<String> cs = toJava(sFuture);
        final CompletableFuture<String> jFuture =
            (CompletableFuture<String>) cs;
        assert (jFuture.get(1000, TimeUnit.MILLISECONDS)
            .equals("Pong"));
    }

    @Test(expected = ExecutionException.class)
    public void shouldReplyToUnknownMessageWithFailure() throws
        Exception {
        Future sFuture = ask(actorRef, "unknown", 1000);
        final CompletionStage<String> cs = toJava(sFuture);
```

```
        final CompletableFuture<String> jFuture =
            (CompletableFuture<String>) cs;
        jFuture.get(1000, TimeUnit.MILLISECONDS);
    }
}
```

上面的 PongActor 测试有两个测试用例，一个针对成功的情况，一个针对失败的情况。

创建 Actor

首先创建一个 ActorSystem，然后通过 actorOf 在刚创建的 Actor 系统中创建一个 Actor，前面的章节对此做过介绍：

```
ActorSystem system = ActorSystem.create();
ActorRef actorRef = system.actorOf(Props.create(JavaPongActor.class));
```

现在向 Actor 询问其对于某个消息的响应：

```
final Future sFuture = ask(actorRef, "Ping", 1000);
```

这一做法相当直接，我们调用 ask 方法，传入以下参数：

- 消息发送至的 Actor 引用；
- 想要发送给 Actor 的消息；
- Future 的超时参数：等待结果多久以后就认为询问失败。

ask 会返回一个 Scala Future，作为响应的占位符。在 Actor 的代码中，Actor 会向 sender()发送回一条消息，这条消息就是在 ask 返回的 Scala Future 中将接收到的响应。

虽然我们无法在 Java 8 中使用 Scala Future，但是可以通过之前导入的库将其转换为 CompletableFuture：

```
final CompletionStage<String> cs = toJava(sFuture);
final CompletableFuture<String> jFuture = (CompletableFuture<String>) cs;
```

我们首先使用 scala.compat.java8.FutureConverters. toJava 对 Scala Future 进行转换，该方法会返回一个 CompletionStage。CompletionStage 是 CompletableFuture 实现的接口，而且这是一个只读的接口。为了调用 get 方法，我们将结果的类型转换为 CompletableFuture。在测试用例外部，我们并不需要进行该转换。

要注意的是，我们在 Future 内存放的数据类型是 String，而 Actor 是无类型的，会返回 Object，因此读者可能会觉得这种无限制的类型转换有问题。当然，在 ActorSystem 外部与 Actor 进行通信的时候需要在这方面多加小心。不过在这条消息中，我们知道 Actor 一定会返回一个 String，所以认为 Future 中存放 String 是安全的。

最后，我们调用 get()方法将测试线程阻塞，并得到结果。在查询失败的例子中，get 方法会抛出一个从 Actor 发送出的 akka.status.Failure 异常。

现在我们就有了一个查询成功的 Future 和一个查询失败的 Future 用来做实验了！

Scala 示例

接下来，我们介绍使用 Scala 的例子。由于 Akka 返回的是 Scala 的 Future，所以使用 Scala 的测试用例更简单一些。

测试用例

下面是完整的 Scala 测试用例。接着我们来详细介绍这个测试：

```
package pong
  //[...imports]
  import akka.pattern.ask
  import scala.concurrent.duration._
  class ScalaAskExamplesTest extends FunSpecLike with Matchers {
    val system = ActorSystem()
    implicit val timeout = Timeout(5 seconds)
    val pongActor = system.actorOf(Props(classOf[ScalaPongActor]))
    describe("Pong actor") {
      it("should respond with Pong") {
        val future = pongActor ? "Ping" //uses the implicit timeout
        val result = Await.result(future.mapTo[String], 1 second)
        assert(result == "Pong")
      }
      it("should fail on unknown message") {
        val future = pongActor ? "unknown"
        intercept[Exception]{
          Await.result(future.mapTo[String], 1 second)
        }
      }
    }
  }
```

创建 Actor

首先创建一个 ActorSystem，然后像前面介绍过的那样，使用 actorOf 在刚创建的 Actor 系统中创建一个 Actor。

为了创建 Future，还要创建一个隐式的 Timeout（注意到我们需要引入 scala.concurrent.duration，然后把一个时间长度传递给 Timeout）：

```
implicit val system = ActorSystem()
implicit val timeout = Timeout(5 seconds)
val pongActor = system.actorOf(Props(classOf[ScalaPongActor]))
```

现在，我们向 Actor 请求一条消息的响应：

```
val future = pongActor ? "Ping"
```

注意

要完成这一操作，我们需要引入 akka.pattern.ask。

请求响应的调用需要引用下面的参数：

- 消息发送对象 PongActor 的 Actor 引用；
- 要发送给 Actor 的消息；
- 隐式传入的 Future 的超时参数：等待结果多久以后认为请求失败。

该调用会返回一个占位符，也就是一个 Future，表示 Actor 返回的响应。在 Actor 的代码中，Actor 会向 sender()返回一个消息，这个消息就是我们会在 Future 中接收到的响应。

最后，我们想在真正接收到结果之前阻塞测试线程。我们使用 Await.result，并传入 Future 和一个超时参数。

```
val result = Await.result(future.mapTo[String], 1 second)
```

Actor 的返回值是没有类型的，因此我们接收到的结果是 Future[AnyRef]。所以应调用 future.mapTo[String]将 Future 的类型转换成我们需要的结果类型。

有了这个例子作为基础，在继续学习本章余下章节的内容时，就可以进行实验，使用了解 Future API 了。

在测试中阻塞进程

我们向 Actor 请求消息会返回一个 Future 作为响应，这展示了从 Actor 系统外部与 Actor 进行通信的方法。在这个测试用例中，我们休眠/阻塞了调用 Await.result 的测试线程，这样就能同步地得到 Future 的结果。

注意

不要在非测试代码中休眠或阻塞线程。

在测试用例中，如果要阻塞 Java 8 的 CompletableFuture，推荐使用的方法是调用 Future 的 get()方法。get()方法会阻塞线程，直到返回结果。

```
jFuture.get().equals("Pong")
```

不过如果没有指定超时参数的话，get()方法是有可能使线程永远休眠的。所以在创建 Scala 的 Future 时，必须要提供超时参数，保证请求超时时 Future 返回失败。

可以使用 scala.concurrent.Await.result 来获取 Scala Future 中的结果。

```
import scala.concurrent.duration._
val result: String = Await.result(future.mapTo[String], 1 second)
```

尽管在 ask 方法中已经设置了 Future 的超时时间，但是在这里仍然必须要提供超时参数。

无论是在 Java 还是 Scala 的例子中，如果 Future 返回失败，那么阻塞线程会抛出异常。Future 失败会产生一个 Throwable，Java 8 的 CompletableFuture 会对这个 Throwable 进行封装并抛出 ExecutionException，而 Scala API 则会直接抛出这个 Throwable。（Scala 没有受检异常，所以可以直接抛出 Throwable，而 Java 会在这里抛出一个非受检的异常类型。）

现在我们就有了一个 Future 的例子，可以基于这个例子来创建测试用例，研究实验结果。由于理解 Future API 对于编写异步代码至关重要，接下来我们就将对其进行深入介绍。极力推荐读者在这些测试用例中对 Future API 做一些研究。为了节省一些篇幅，所有的例子都可以在网上的第 2 章 Actor 与并发的代码中找到。

2.5.3 理解 Future 和 Promise

现代化的 Future 隐式地处理了两种情况：失败与延迟。要了解如何把阻塞式 IO 转化成非阻塞式 IO，我们必须学习一些不同的表示失败处理和延迟处理的抽象概念。刚开始可能会显得有点困难，但是一旦真正理解了，大多数开发者就能够习惯这种编程范式了。

Future——在类型中表达失败与延迟

像 ask 模式这样的异步 API 会返回一个占位符，类似前面提到的 Future 类型。我们可以了解如何使用不同的方法在测试用例中与 PongActor 进行交互，以及如何让代码变得越来越简洁。强烈推荐读者在前面创建的测试用例的基础上学习本小节余下的内容。

准备 Java 示例

首先，在 Java 8 的例子中，为了避免冗余，我们将 ask 方法简化，并封装到一个方法中。这样看上去就像一个真正的异步 API 了：

```
public CompletionStage<String> askPong(String message){
    Future sFuture = ask(actorRef, "Ping", 1000);
    CompletionStage<String> cs = toJava(sFuture);
    return cs;
}
```

接着我们就可以来创建简单的测试用例：

```
@Test public void printToConsole() throws Exception {
    askPong("Ping").
        thenAccept(x -> System.out.println("replied with: " + x));
    Thread.sleep(100);
}
```

准备 Scala 示例

首先定义一个简单的方法，去除冗余，稍稍增加示例的可读性：

```
def askPong(message: String): Future[String] = (pongActor ? message).mapTo[String]
```

接着使用运行在多线程上的异步操作，因此需要引入隐式的 ExecutionContext。可以创建如下的测试用例并进行实验：

```
describe("FutureExamples"){
  import scala.concurrent.ExecutionContext.Implicits.global
  it("should print to console"){
    (pongActor ? "Ping").onSuccess({
      case x: String => println("replied with: " + x)
    })
    Thread.sleep(100)
  }
}
```

关于休眠

这个测试并没有进行任何断言，但是已经展示了真正的异步行为。我们可以通过观察运行效果来确认异步操作是否成功（在这个测试用例中，我们希望打印到控制台）。如果希望事件异步发生的话，我们可能时不时地需要将测试线程休眠。和阻塞一样，在测试中休眠线程是没问题的，不过在任何时候都不应该在真正的代码中休眠线程。

尽管这些测试并不进行真实的测试，但是它们对于帮助我们观察异步操作实验的效果是很有用的。在花一些时间理解 Future 之后，我们将学习如何在异步代码中进行断言。

剖析 Future

Future[T]/CompletableFuture<T>成功时会返回一个类型为 T 的值，失败时则会返回 Throwable。我们将分别学习如何处理这两种情况（成功与失败），以及如何将 Future 的值转换成有用的结果。

成功情况的处理

在测试中已经看到，PongActor 会在接收到“Ping”时返回“Pong”。我们将使用这个例子来展示与 Future 进行交互的不同方法。

对返回结果执行代码

有时候我们需要使用返回结果做一些“简单的事情”。可能是将事件记录到日志，也可能是通过网络返回一个响应。我们可以“注册”事件，一旦 Future 的结果返回就执行这个事件。

就像上面的例子中那样，在 Java 8 中，可以使用 thenAccept 来操作返回结果。

```
askPong("Ping").thenAccept(x -> System.out.println("replied with: " + x));
```

而在 Scala 中，可以使用 onSuccess：

```
(pongActor ? "Ping").onSuccess({
  case x: String => println("replied with: " + x)
})
```

注意

注意到 onSuccess 接受一个部分函数作为参数，所以非常适合用来处理 Akka 的无类型响应（通过模式匹配来判断返回结果的类型）。

对返回结果进行转换

最常见的一种用例就是在处理响应之前先异步地对其进行转换。例如，我们可能需要从数据库获取数据，然后将其转换成一个 HTTP 响应，再返回给客户端。

在大多数 API 中，类型转换可以通过 map 来完成，例如 Scala 的 Future。

```
askPong("Ping").map(x => x.charAt(0))
```

在 Java 8 中，我们调用 thenApply：

```
askPong("Ping").thenApply(x -> x.charAt(0))
```

上面的操作会返回一个新的 Future，包含 Char 类型。我们可以在对返回结果进行转换后将新得到的 Future 再传递给其他方法，做进一步处理。

对返回结果进行异步转换

如果需要进行异步调用，那么首先要对返回结果进行另一个异步调用，这样代码就会看上去有一点乱：

```
//Java
CompletionStage<CompletionStage<String>> futureFuture =
    askPong("Ping").thenApply(x -> askPong(x));
//Scala
val futureFuture: Future[Future[String]] =
askPong("Ping").map(x => {
  askPong(x)
})
```

很多情况下，需要进行一个异步调用，然后像上面的例子中一样，在得到结果后进行另一个异步调用。不过这样一来，结果就会嵌套在两层 Future 中了。这种情况是很难处理的，要将结果扁平化，使得结果只在一个 Future 中，我们需要的是一个 Future[String]/CompletionStage[String]。

有很多方法都可以用来做这样的链式异步操作。在 Java 中使用 thenCompose：

```
CompletionStage<String> cs = askPong("Ping").thenCompose(x ->
askPong("Ping"));
```

不出意料地，在 Scala 中使用 flatMap：

```
val f: Future[String] = askPong("Ping").flatMap(x => askPong("Ping"))
```

一旦对第一个“Ping”做出了响应，就发送第二个“Ping”并返回包含结果值的 Future 作为响应。

注意到我们可以继续像这样把异步操作连接到一起。这是一种进行流数据处理的很强大的方法。我们可以向一个远程服务发起调用，然后使用得到的结果向另一个服务发起调用。

其中任何一个调用失败都会导致整个 Future 失败。接下来我们就来看一下失败的情况。

失败情况的处理

失败情况是有可能发生的，而我们也需要去处理这些失败情况。所有的失败情况最终都会由一个 Throwable 来表示。和成功的情况类似，有许多方法可以帮助我们来处理失败情况，甚至是从失败中恢复。

2.5.4 在失败情况下执行代码

很多时候，我们都想要在失败情况下做些什么。最基本的就是在失败情况下向日志中打印一些信息。

在 Scala 中，有一种很简单的方法支持这种需求：onFailure。这个方法接受一个部分函数作为参数，而这个部分函数接受一个 Throwable。

```
askPong("causeError").onFailure {
  case e: Exception => println("Got exception")
}
```

不幸的是，在 Java 8 中，没有面向用户的用于失败处理的方法，因此我们在这里引入 handle()来处理这种情况：

```
askPong("cause error").handle((x, t) -> {
    if(t != null){
        System.out.println("Error: " + t);
    }
    return null;
});
```

handle 接受一个 BiFunction 作为参数，该函数会对成功或失败情况进行转换。handle 中的函数在成功情况下会提供结果，在失败情况下则会提供 Throwable，因此需要检查 Throwable 是否存在（结果和 Throwable 中只有一个不是 null）。如果 Throwable 存在，就向日志输出一条语句。由于我们需要在该函数中返回一个值，而失败情况下又不需要对返回值做任何操作，因此直接返回 null。

2.5.5 从失败中恢复

很多时候，在发生错误的时候我们仍然想要使用某个结果值。如果想要从错误中恢复的话，可以对该 Future 进行转换，使之包含一个成功的结果值。

在 Java 中，可以使用 exceptionally 将 Throwable 转换为一个可用的值。

```
CompletionStage<String> cs = askPong("cause error")
    .exceptionally(t -> {
        return "default";
    });
```

在 Scala 中，有一个 recover 方法提供相同的功能。同样地，recover 方法也接受一个 PartialFunction 作为参数，所以我们可以对异常的类型进行模式匹配：

```
val f = askPong("causeError").recover {
    case t: Exception => "default"
}
```

2.5.6　异步地从失败中恢复

我们经常需要在发生错误时使用另一个异步方法来恢复，下面是两个用例。

- 重试某个失败的操作。
- 没有命中缓存时，需要调用另一个服务的操作。

下面展示了一个重试操作：

```
askPong("cause error")
    .handle( (pong, ex) -> ex == null
        ? CompletableFuture.completedFuture(pong)
        : askPong("Ping")
    ).thenCompose(x -> x);
```

我们需要分两步来完成这一操作。首先，检查 exception 是否为 null。如果为 null，就返回包含结果的 Future，否则返回重试的 Future。接着，调用 thenCompose 将 CompletionStage[CompletionStage[String]]扁平化。

在 Scala 中，我们要调用的函数是 recoverWith：类似专门用于错误情况的 flatMap，所以要比 Java 的处理方法可读性更高，也更简洁得多。

```
askPong("causeError").recoverWith({
    case t: Exception => askPong("Ping")
})
```

构造 Future

很多时候，我们需要执行多个操作，而且可能想要在代码库的不同位置来执行这些操作。之前介绍到的每个方法调用都会返回一个新的 Future，而我们又可以对这个新的 Future 执行其他操作。

2.5.7 链式操作

我们已经介绍了 Future 的基本使用方法。应用函数式风格来处理延迟和失败的好处之一就是可以把多个操作组合起来，而在组合的过程中无需处理异常。我们可以把注意力放在成功的情况上，在链式操作的结尾再收集错误。

之前介绍的每个用于结果转换的方法都会返回一个新的 Future，可以处理这个 Future，也可以将其与更多操作链接到一起。

总结一下，执行多个操作时，我们最后使用一个恢复函数来处理所有可能发生的错误。可以用我们想要的任何顺序来组合这些函数（combinators）来完成我们需要完成的工作。

在 Java 中：

```
askPong("Ping")
    .thenCompose(x -> askPong("Ping" + x))
    .handle((x, t) -> {
        if(t != null) {
            return "default";
        } else {
            return x;
        }
    });
```

在 Scala 中：

```
val f = askPong("Ping")
  .flatMap(x => askPong("Ping" + x))
  .recover({ case Exception => "There was an error" })
```

在上面的例子中，我们得到了一个 Future，然后调用 thenCompose/flatMap，在第一个操作完成时异步地发起另一个调用。接着，在发生错误时，我们使用一个 String 值来恢复错误，保证 Future 能够返回成功。

在执行操作链中的任一操作时发生的错误都可以作为链的末端发生的错误来处理。这样就形成了一个很有效的操作管道，无论是哪个操作导致了错误，都可以在最后来处理异常。我们可以集中注意力描述成功的情况，无需在链的中间做额外的错误检查。可以在最后单独处理错误。

2.5.8 组合 Future

我们经常需要访问执行的多个 Future。同样有很多方法可以用来处理这些情况。在

Java 中，可以使用 CompletableFuture 的 thenCompose 方法，在 Future 的值可用时访问到这些值：

```
askPong("Ping")
    .thenCombine(askPong("Ping"), (a,b) -> {
        return a + b; //"PongPong"
    });
```

在 Scala 中，也可以使用 for 推导式将多个 Future 组合起来。我们能够像处理任何其他集合一样，解析出两个 Future 的结果并对它们进行处理。（要注意的是，这只不过是 flatMap 的一个“语法糖”：相比于 flatMap，我更喜欢这个语法。）

```
val f1 = Future {4}
val f2 = Future {5}

val futureAddition: Future[Int] =
  for (
    res1 <- f1;
    res2 <- f2
  ) yield res1 + res2
```

这个例子展示了一种处理多个不同类型 Future 的机制。通过这种方法，可以并行地执行任务，同时处理多个请求，更快地将响应返回给用户。这种对并行的使用可以帮助我们提高系统的响应速度。

2.5.9　处理 Future 列表

如果想要对集合中的每个元素执行异步方法，那么可以使用 Future 列表。

例如，在 Scala 中，如果我们有一个消息列表，对于列表中的每个消息，向 PongActor 发送查询，最后会得到如下的一个 Future 列表：

```
val listOfFutures: List[Future[String]] = List("Pong", "Pong",
"failed").map(x => askPong(x))
```

对 Future 列表的处理并不容易。我们希望得到的是一个结果列表，也就是要反转一下类型，把 List[Future]转换成 Future[List]。Future 的 sequence 方法就是用来完成这一工作的：

```
val futureOfList: Future[List[String]] = Future.sequence(listOfFutures)
```

现在我们就有了一个可以使用的类型。例如，如果在 futureOfList 上调用 map 方法

的话，就可以得到一个 List[String]，这就是我们想要的结果类型。不过这里有一个问题。一旦 Future 列表中的任何一个 Future 返回失败，那么 sequence 生成的 Future 也会返回失败。如果不希望这种情况发生，想要得到一些成功的结果值，那么可以在执行 sequence 之前将返回失败的 Future 逐一恢复：

```
Future.sequence(listOfFutures.map(future => future.recover {
  case Exception => ""
}))
```

在 Java 8 的核心库中，并没有提供具备类似功能的方法。不过我们能找到一些代码示例用于实现类似的功能。

2.5.10 Future 速查表

表 2-1 给出了本章中介绍过的一些基本操作：

表 2-1

操作	Scala Future	Java CompletableFuture
Transform Value	.map(x => y)	.thenApply(x -> y)
Transform Value Async	.flatMap(x => futureOfY)	.thenCompose(x -> futureOfY)
Return Value if Error	.recover(t => y	.exceptionally(t -> y)
Return Value Async if Error	.recoverWith(t => futureOfY)	.handle(t,x -> futureOfY).thenCompose(x->x)

本小节对 Future 和 Promise 的 API 做了比较详细的介绍。理解这些抽象概念是很必要的，因此推荐读者实际编写一些异步的代码，使用一下 Future，把这个知识点掌握得更牢固一些。在下一小节中，我们将介绍如何通过返回 Future 来和 Actor 进行交互。读者应该花点时间练习本小节的内容，了解如何处理多个 Future 结果，确保有一个扎实的基础，帮助后面的学习。

构建分布式系统——AkkademyDb 和客户端

本书的目的是教授如何构建分布式应用程序，因此我们要把本章中介绍过的所有知识点结合起来，编写一个小型分布式应用程序。尽管代码相当简单，但是这个例子直接展示了两个远程系统如何通过 Akka 相互通信，虽然从结构上来看还是比较先进的，不过我认为还是要马上展示一下 Akka 的强大之处，让读者保持对本书的兴趣。如果对 Akka 能提供的强大功能有些许了解，就会很有兴趣继续阅读余下的章节了。

我们将构建一个服务和一个客户端。也就是数据库和与之通信的数据库客户端。要

通过网络在客户端和服务之间发送消息，我们的两个项目需要共享相同的消息。

我们也可以在两个项目中都包含消息，不过为了让例子更简短一些，我们把消息放在服务器项目中，然后在客户端项目中导入服务器项目（也就包含了消息）。

首先，我们将扩展第一章中的服务器项目，定义希望数据库接受的所有消息。接着，我们将针对这些消息分别实现数据库中的相关功能。

在构建了这些基本操作之后，我们将编写一个 main()方法来运行数据库。启动应用程序后，我们将构建一个 ActorSystem 以及一个在该 ActorSystem 中的 Actor，这就构成了我们的第一个 Akka 微服务。

我们还将创建一个数据库客户端，用于展示如何请求服务器，以及如何从远程 Actor 中获取 Future。服务器端的服务接收到客户端的请求后将返回 Future。这样我们就已经编写了一个可以使用的键值存储数据库（和 redis 很类似）以及一个可以使用该数据库的远程客户端，如图 2-4 所示。

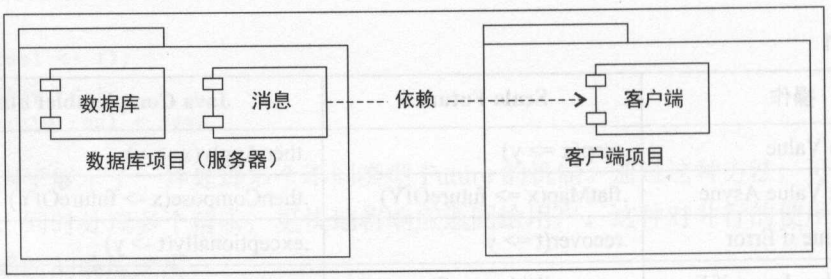


图 2-4

2.5.11　准备数据库与消息

首先，我们要构造几种消息。

- Get 消息：如果 key 存在，就返回 value；
- Key Not Found 异常消息：如果 key 不存在，就返回该异常；
- Set 消息：设置某个键值对，返回状态。

我们将在服务器端实现这些消息及其行为，以及用于启动该数据库的 main 函数。需要注意的是，我们将使用“第 1 章：初识 Actor”中的项目，并在这基础上添加本章介绍的功能，比如返回响应及失败情况的处理。

消息

由于我们将在通过网络连接的独立应用程序之间远程发送消息，因此需要能够对所有的消息进行序列化，这样 Akka 就能够将表示这些消息的对象转换成能够在网络应用程序之间传输的表示形式。我们将实现 SetRequest，GetRequest 和 KeyNotFound-

Exception。

使用 Java 实现的消息：

```
public class SetRequest implements Serializable {
    public final String key;
    public final Object value;
    public SetRequest(String key, Object value) {
        this.key = key;
        this.value = value;
    }
}
public class GetRequest implements Serializable {
    public final String key;
    public GetRequest(String key) {
        this.key = key;
    }
}
public class KeyNotFoundException extends Exception implements
Serializable {
    public final String key;
    public KeyNotFoundException(String key) {
        this.key = key;
    }
}
```

使用 Scala 实现的消息：

```
case class SetRequest(key: String, value: Object)
case class GetRequest(key: String)
case class KeyNotFoundException(key: String) extends Exception
```

这些都是很简单的类。由于消息是不可变的，所以我们没有使用 Java 的 getter 方法，而是直接将成员变量设为 public。当然如果读者想要添加 getter 方法的话也可以。

注意

消息始终都应该是不可变的。

Scala 的 case class 是可以被序列化的。

实现数据库功能

前文已经介绍过如何使用 sender() tell/! 来返回响应消息，以及如何返回 Status.Failure(Exception)。接下来，我们就将实现对所有消息的响应，并且在 GetRequest

请求获取的键值不包含在键值存储中时返回失败响应。

下面是 Java 的 receive 语句：

```
receive(ReceiveBuilder
    .match(SetRequest.class, message -> {
        log.info("Received Set request: {}", message);
        map.put(message.key, message.value);
        sender().tell(new Status.Success(message.key), self());
    })
    .match(GetRequest.class, message -> {
        log.info("Received Get request: {}", message);
        String value = map.get(message.key);
        Object response = (value != null)
            ? value
            : new Status.Failure(new KeyNotFoundException(message.key));
        sender().tell(response, self());
    })
    .matchAny(o ->
        sender().tell(new Status.Failure(new ClassNotFoundException()), self())
    )
    .build()
);
```

下面是 Scala 的 receive 方法：

```
override def receive = {
  case SetRequest(key, value) =>
    log.info("received SetRequest - key: {} value: {}", key, value)
    map.put(key, value)
    sender() ! Status.Success
  case GetRequest(key) =>
    log.info("received GetRequest - key: {}", key)
    val response: Option[String] = map.get(key)
    response match{
      case Some(x) => sender() ! x
      case None => sender() ! Status.Failure(new KeyNotFoundException(key))
    }
  case o => Status.Failure(new ClassNotFoundException)
}
```

除了使用的语言不同之外，上面两者基本上等价的。如果 Actor 接收到一个 SetRequest，就将键值存储到 map 中。在第 1 章中，只是返回了一个 Success 消息，这里对此做了更新。如果接收到的是 GetRequest，Actor 就会尝试从 map 中获取结果。如果

找到 key，就将 value 返回。如果没有找到，就返回一个包含 KeyNotFoundException 的失败消息。最后，我们也更新了第 1 章中接收到未知消息时的行为，会返回一个包含了 ClassNotFoundException 的错误消息，不过读者也可以提供一个自定义的异常，表达更清晰的异常语义。

支持远程

我们需要支持远程应用程序通过网络远程访问上面定义的 Actor。幸运的是，这是件很简单的事。我们只需要在 build.sbt 中添加用于远程访问的依赖即可：

```
"com.typesafe.akka" %% "akka-remote" % "2.3.6"
```

接着，只要添加配置文件，就可以启用 Actor 的远程访问功能了。在 src/main/resources 文件夹中新建一个文件，命名为 application.conf，然后把下面的配置内容添加到该文件中，其中包含了要监听的主机和端口。Akka 负责解析 application.conf。这是一个 HOCON 文件，是一种类型安全的配置文件，格式和 JSON 类似，与其他配置文件格式一样，使用方便。Akka 文档中经常提到该格式的配置文件，笔者认为如果要配置多个属性，HOCON 是一种相当不错的用于配置文件的格式。要注意的是，如果将其命名为 application.properties 并且使用属性配置文件的格式（比如 keypath.key=value），Akka 也是可以解析的。下面是 application.conf 的内容：

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = 2552
    }
  }
}
```

Main 函数

最后，我们需要为数据库添加一个 main 方法，启动 Actor 系统，并创建 Actor。

在 Java 中，我们将添加一个类：com.akkademy.Main：

```
public class Main {
    public static void main(String... args) {
```

```
        ActorSystem system = ActorSystem.create("akkademy");
        system.actorOf(Props.create(AkkademyDb.class), "akkademy-db");
    }
}
```

在 Scala 中，可以把 Main 对象放在 com.akkademy.AkkademyDb.scala 文件中：

```
object Main extends App {
  val system = ActorSystem("akkademy")
  system.actorOf(Props[AkkademyDb], name = "akkademy-db")
}
```

我们只需要创建 ActorSystem，然后在该 Actor 系统中创建一个 Actor。注意到我们给 Actor 指定了一个名字：akkademy-db。有了这个名字，客户端就能够很容易地查询到 Actor。同时由于 Akka 会在发生错误时把 Actor 的名字记录到日志中，指定名字也使得调试变得更容易。

发布消息

现在我们就要在本地发布这些消息，这样就能够在客户端项目中使用它们了。如果想要发布到 Nexus 或是 Artifactory，那么需要在 build.sbt 中设置仓库信息。不过在这个例子中，我们直接将消息发布到本地。

我们需要在 build.sbt 文件中添加机构和版本信息，如下所示：

```
name := "akkademy-db"
organization := "com.akkademy-db"
version := "0.0.1-SNAPSHOT"
```

“-SNAPSHOT”表示该版本还不稳定，有可能发生改变。由于可能会重新发布服务器，所以应该将这个标签添加到版本信息中。如果准备正式发布代码，就可以从版本信息中删除“-SNAPSHOT”标签，表示这是一个稳定版本。

最后，需要在发布的内容中排除 application.conf，防止客户端也能够试图启动远程服务器。当然，更好的做法是把消息放在一个单独的库中，这里只是为了简单才这么做。把下面的内容添加到 build.sbt 文件中，就可以在发布时排除 application.conf：

```
mappings in (Compile, packageBin) ~= { _.filterNot { case (_, name) =>
  Seq("application.conf").contains(name)
}}
```

如果把消息放在一个单独的库中（当然可以），就不需要在 build.sbt 添加上面的配置将 application.conf 排除了。现在就已经完成了构建文件的配置。可以在命令行中进入项目的根目录，直接运行 activator 的 publish-local 任务，发布项目：

```
$activator publish-local
```

启动数据库

接下来我们将构建客户端，并且通过编写一些集成测试来展示如何将客户端与服务器进行集成，因此需要服务器保持运行。现在启动数据库：

```
$activator run
```

Akka 会输出日志，表明其正在监听远程连接，并且给出当前服务器的地址（我们马上会在客户端中使用该地址）：

```
[Remoting] Remoting now listens on addresses: [akka.tcp://
akkademy@127.0.0.1:2552]
```

2.5.12 编写客户端

我们已经发布了消息，键值数据库也已经处于运行中。现在就可以编写一个客户端，连接并使用服务器提供的服务。我们将以此来结束第一个分布式应用程序。

项目结构

首先需要为客户端创建一个项目，并且导入服务器项目，得到消息的定义。我们将像“第 1 章：初始 Actor”中一样创建项目结构，如果需要的话读者可以重新阅读相关的内容。运行 activator-new，选择 minimal-java 或是 minimal-akka 项目。将该项目命名为 akkademy-db-client。

修改 build.sbt

我们需要在 build.sbt 文件中添加项目需要的依赖。在新建的项目中，在 build.sbt 中添加消息定义所需的下述依赖：

```
"com.akkademy-db"   %% "akkademy-db"      % "0.0.1-SNAPSHOT"
```

除了新建项目后已经默认添加的测试框架依赖之外，这个依赖包含了 Scala 项目所需要的所有依赖项。

在 Java 项目中，还需要添加一个 scala-java8-compat 库，用于将 Actor 生成的 Scala Future 转换成 Java 的 CompletionStage：

```
"org.scala-lang.modules" %% "scala-java8-compat" % "0.6.0"
```

构建客户端

在这一小节中，我们将构建客户端，连接远程 Actor，然后分别实现用于处理 SetRequest 和 GetRequest 消息的方法。

首先，将 Java 代码放在 com.akkademy.JClient 中：

```
public class JClient {
    private final ActorSystem system = ActorSystem
            .create("LocalSystem");
    private final ActorSelection remoteDb;
    public JClient(String remoteAddress) {
        remoteDb = system.actorSelection("akka.tcp://akkademy@" +
            remoteAddress + "/user/akkademy-db");
    }
    public CompletionStage set(String key, Object value) {
        return toJava(ask(remoteDb, new SetRequest(key, value),
            2000));
    }
    public CompletionStage<Object> get(String key){
        return toJava(ask(remoteDb, new GetRequest(key), 2000));
    }
}
```

将 Scala 代码放在 com.akkademy.SClient 中：

```
class SClient(remoteAddress: String) {
  private implicit val timeout = Timeout(2 seconds)
  private implicit val system = ActorSystem("LocalSystem")
  private val remoteDb = system.actorSelection(
    s"akka.tcp://akkademy@$remoteAddress/user/akkademy-db")
  def set(key: String, value: Object) = {
    remoteDb ? SetRequest(key, value)
  }
  def get(key: String) = {
    remoteDb ? GetRequest(key)
  }
}
```

代码相当简单。首先创建一个本地的 ActorSystem，然后通过构造函数中提供的地址得到指向远程 Actor 的引用。接着，分别为 GetRequest 和 SetRequest 两种消息创建方法。我们向远程 Actor 发送本项目中导入的消息，然后得到返回的 Future。注意到我

们在发起的请求中随意使用了一个超时参数值。理想情况下，这个超时参数最好是可以配置的。

在 Java 代码中，我们将 scala.concurrent.Future 转换成 Completion Stage，然后返回 CompletionStage。这样能够给库的用户提供一个更好的 Java API。

接下来，我们将编写一个简单的测试用例来进行集成测试。

测试

由于要编写的是集成测试，所以需要数据库服务器保持运行。在这个例子中，我们将在远程数据库中创建一条记录，然后获取该记录。

Java 的例子如下：

```
public class JClientIntegrationTest {
    JClient client = new JClient("127.0.0.1:2552");
    @Test
    public void itShouldSetRecord() throws Exception {
        client.set("123", 123);
        Integer result = (Integer) ((CompletableFuture) client.
get("123")).get();
        assert(result == 123);
    }
}
```

Scala 的例子如下：

```
class SClientIntegrationSpec extends FunSpecLike with Matchers {
  val client = new SClient("127.0.0.1:2552")
  describe("akkademyDbClient") {
    it("should set a value") {
      client.set("123", new Integer(123))
      val futureResult = client.get("123")
      val result = Await.result(futureResult, 10 seconds)
      result should equal(123)
    }
  }
}
```

在测试我们编写的 API 时，需要用到在介绍 Future 时学到的知识。由于这只是个测试用例，所以使用了 Await 和 get。虽然我们仅仅学完了“第 2 章：Actor 与并发”，但是现在已经完全可以证明，使用 Akka 来编写分布式应用程序是切实可行的。

2.6　课后作业

由于本章介绍了使用 Akka 所需的核心技能，因此完成本章的作业非常重要。本章的示例代码虽然简单，但是提前介绍了远程 Actor 的连接。在接下来的几章中，我们将只使用本地 Actor 系统，所以示例会更加简单。不过通过运行本章的示例程序，读者能够对 Akka 试图解决的问题有一些认识。所以推荐读者花上些许时间学习一下本章的例子。本章的内容（尤其是 Future API）是接下来所有内容的基础。我们需要在学习了本章的知识后为后面的章节打下扎实的基础。

2.6.1　基本知识

请完成下面的任务，确保理解如何使用本章介绍的内容进行编程。

- 编写一个小型的服务，其中包含一个负责将字符串翻转的 Actor。当接收到未知类型的消息时直接报错。
- 编写一个服务，使用 Java 或者 Scala API 完成功能接口，并返回 Future。
- 编写测试用例，覆盖成功和错误情况。
- 编写一个测试，向 Actor 发送一个字符串列表，并且验证所有的结果。请使用 Sequence。（可以在 http://www.nurkiewicz.com/2013/05/java-8- completablefuture-in-action.html 处找到 Java 8 的 Sequence 实现。）

2.6.2　项目作业

对示例代码进行扩展。

- 向 AkkademyDb 应用程序中添加 SetIfNotExists 和 Delete 消息，对这两种消息的处理均为原子操作。
- 如果想要练习一下 sbt 的使用，可以把消息的定义移到一个单独的项目中并发布。
- 完成客户端的测试。我们还没有编写对于失败情况的测试，比如接收到表示 key 不存在的响应消息。请完成该测试。
- 开始编写读者自己的的项目。
  - 编写一个 Actor，将项目中的某些功能开放出去：接收消息，完成任务，并返回响应。
  - 编写测试用例，向 Actor 发送请求，覆盖成功和错误的情况。

2.7 小结

本章介绍了使用 Akka 编写应用程序的基础。现在我们已经有了开始真正编写分布式应用程序所需的所有预备知识了。我们深入分析了 Actor 代码：创建一个 Actor 实例、向 Actor 发起查询、在 Actor 中返回消息的响应、从 ActorSystem 外部获取 Actor 返回的响应以及 Future 的使用。

现在，我们已经有了足够的知识将消息驱动的异步 Akka 应用程序提供给别人访问（可以使用 Akka 编写库和服务，并利用 Scala 核心库和 Java 8 的 Future API 返回结果）。这就是下一章要介绍的内容。

第 3 章
传递消息

本章介绍 Akka 中消息传递机制的所有细节。我们将了解不同的消息模式，也就是在不同 Actor 之间传递消息的不同方法。我们也会介绍消息传递的调度方法（延迟或重复消息的传递）。本章将阐述如何使用这些消息模式将多个 Actor 串联起来，完成工作。

为了展示在多个 Actor 之间传递消息的所有方法，我们将编写一个新的示例服务，作为 AkkademyDb 的另一个客户端。该服务是一个简单的用于解析文章的组件。

本章将涉及消息处理的一些基本机制：

- 将消息定义为不可变；
- 向 Actor 请求响应；
- 转发消息；
- Future 管道。

3.1 示例问题

假设我们所在的团队需要开发一款移动新闻阅读器。部分团队成员将负责开发这款运行在移动设备上的应用程序。这款阅读器将从主要的新闻网站获取 RSS 摘要，组合起来后将最新的文章推送并展示给用户。当用户选择一篇想要阅读的文章时，阅读器将会显示简单的文本内容，并专门针对移动设备上的阅读体验进行优化。

我们需要编写一个服务，接收文章的 URL，返回文章的文本消息体。返回结果只包含消息体，不包含 HTML 标签。因为会有许多用户阅读这些新文章，所以阅读器应该把所有已经解析过的文章缓存起来，这样就可以将阅读过的文章快速返回给用户。文章的缓存需要保存在运行 AkkademyDb 实例的另一台机器上。

需求概述：

- 开放一个 HTTP 端口，接收文章的 URL；

- 返回消息的主文本消息体；
- 在 AkkademyDb 中保存文章的缓存。

3.1 项目结构

我们仍将使用 activator 来新建另一个项目。可以参见“第 1 章：初始 Actor”了解更多细节。

执行下述步骤：

1. 运行：activator run；
2. 选择 minimal-java 或 minimal-scala 项目；
3. 给项目取名为 akkademaid；
4. 向 build.sbt 中添加 akkademy-db 和 boilerpipe 的依赖：

```
libraryDependencies ++= Seq(
    "com.syncthemall" % "boilerpipe" % "1.2.2",
    "com.akkademy-db" %% "akkademy-db-scala" % "0.0.1-SNAPSHOT",
    "com.syncthemail" % "boilerpipe" % "1.2.2"
)
```

添加了 akkademy-db 依赖之后，就可以访问与之通信所需的消息定义了。而添加 Java 库 boilerpipe 是为了满足从网页获取文本体的需求。build.sbt 中默认就会包含所需的测试框架。

核心功能

boilerpipe 库提供了文章解析的功能。我们只需要调用 ArticleExtractor.getInstance.getText(input)即可，传入的参数是一个 Stream 或是 String。

在学习解决这个问题的不同方法的过程中，我们将反复使用到这个例子和 AkkademyDb。

3.2 消息传递

这一小节将介绍向 Actor 发送消息的各种方法。除了介绍核心的消息模式之外，还会介绍调度机制。

有 4 种核心的 Actor 消息模式：tell、ask、forward 和 pipe。我们已经了解过 tell 和 ask，不过 sender()都不是 Actor。在这里，将从 Actor 之间发送消息的角度来介绍所有关

于消息传递的概念。

- **Ask**：向 Actor 发送一条消息，返回一个 Future。当 Actor 返回响应时，会完成 Future。不会向消息发送者的邮箱返回任何消息。
- **Tell**。向 Actor 发送一条消息。所有发送至 sender()的响应都会返回给发送消息的 Actor。
- **Forward**：将接收到的消息再发送给另一个 Actor。所有发送至 sender()的响应都会返回给原始消息的发送者。
- **Pipe**：用于将 Future 的结果返回给 sender()或另一个 Actor。如果正在使用 Ask 或是处理一个 Future，那么使用 Pipe 可以正确地返回 Future 的结果。

在接下来的小节中，我们将介绍这些模式以及 Akka 中消息传递的基本原则。

3.2.1　消息是不可变的

前面提到过，消息应该是不可变的。由于 Akka 基于 JVM，而 Java 和 Scala 都支持可变类型，所以是有可能发送可变消息的。不过如果这么做，就可能会失去 Akka 在消除共享状态方面提供的诸多益处。一旦有了可变消息，就引入了一种风险：开发者可能会在某些时候开始修改消息，而他们修改消息的方式可能会破坏应用程序的运行。当然，如果不修改消息的状态，那么要安全地使用可变消息也不是不可能。不过最好还是使用不可变消息，确保不会因为未来的变化而引入错误。

有两种方法可以用来定义可变消息：可变引用以及可变类型。最近我见到了 Jamie Allen，他通过一个矩阵来说明了可变引用、可变类型和消息之间的关系，如表 3-1 所示。

表 3-1

引用	可变类型	不可变类型
可变引用	☹☹	☹
不可变引用	☹	☺

引用和类型均为可变的情况是最糟糕的。下面是一个例子：

```
public class Message {
    public StringBuffer mutableBuffer;
    public Message(StringBuffer: mutableBuffer) {
        this.mutableBuffer = mutableBuffer;
    }
}
```

```
class Message(var mutableBuffer: StringBuffer = new StringBuffer);
```

在这个例子中，mutableBuffer 是一个引用，可以指向另一个对象，其指向对象本身的状态也可以改变，也就是上面矩阵中最糟糕的情况。

消息中引用的变量（比如成员变量）既可以是可变的，也可以是不可变的。在 Scala 中标记为 val 的成员变量是不可变引用，而标记为 var 的成员变量是可变的，可以指向另一个对象或基本类型。在 Java 中，标记为 final 的成员变量是不可变引用，没有标记为 final 的则为可变引用。消息中任何成员变量的可变性一定属于上述矩阵中的一种。要支持不可变消息，就必须将成员变量作为构造函数参数传入。

下面展示了如何修改引用：

```
Message message = new Message(new StringBuffer("original"));
message.mutableBuffer = new StringBuffer("new");

val message = new Message(new StringBuffer("original"))
message.mutableBuffer = new StringBuffer("new")
```

这个例子新建了一条消息，然后修改 mutableBuffer 引用，使之指向另一个新建的 StringBuffer。消息创建时传入的是 StringBuffer ("original")，后来被修改成了 StringBuffer("new")。这就是通过修改引用来修改消息的方法，如图 3-1 所示。

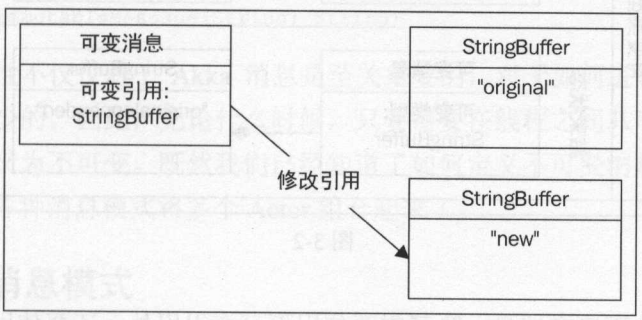


图 3-1

我们希望防止人为修改消息中的引用，所以需要将引用标记为 val/final。下文在上面例子的基础上进行改进，在消息中使用不可变引用：

```
public class Message {
    public final StringBuffer mutableBuffer;

    Message(StringBuffer mutableBuffer) {
        this.mutableBuffer = mutableBuffer;
    }
}
```

在 Scala 中，如果没有在声明中给出任何访问修饰符，成员变量的引用默认就是不可变的 val。在 Java 中，我们使用 final 关键字将引用设置为不可变。现在，就无法改变 mutableBuffer 引用，使之指向另一个新对象了：我们可以保证该消息将永远指向相同的 StringBuffer。不过还有一个问题：由于 StringBuffer 是可变的，因此可以修改 StringBuffer 对象本身。可以在引用末尾添加新字符。如果多个 Actor 都引用了这条消息，那么就可能会发生并发错误。下面是修改 StringBuffer 的一个例子：

```
Message message = new Message(new StringBuffer())
message.mutableBuffer.append("appended");
val message = new Message(StringBuffer("original"))
message.mutableBuffer.append("appended")
```

这就说明了即使消息的 mutableBuffer 引用是不可变的，但是由于 StringBuffer 是可变类型，所以内存中的 StringBuffer 仍然可以被修改。如图 3-2 所示。

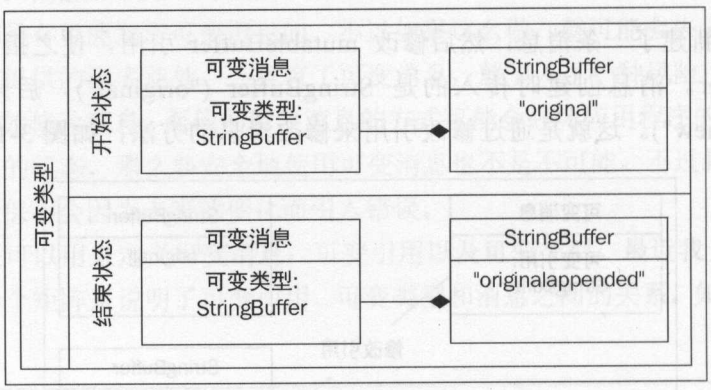


图 3-2

要定义真正的不可变消息，除了需要使用不可变引用外，还要使用不可变类型。由于 String 是不可变类型，因此可以通过使用 String 代替 StringBuffer 使得消息不可变。下面是理想的不可变消息的例子：

```
public class ImmutableMessage{
    public final String immutableType;
    public ImmutableMessage(String immutableType) {
        this.immutableType = immutableType;
    }
```

```
}
class ImmutableMessage(immutableType: String)
```

现在我们就无法通过任何方式来修改消息了。这个消息是线程安全的，可以在多个线程或机器之间发送该消息，无需担心该消息在其生命周期中会以某种方式被修改：

```
new ImmutableMessage("can't be changed");
```

还可以做一个改进：在 Scala 中使用 case class 代替 class。推荐使用 case class 是因为它会自动生成一些有用的成员方法，比如默认的 toString()和 copy()方法。case class 也可以用于 Akka 远程访问的序列化。在 Java 中，由于我们可能希望在互联网上发送消息，因此也经常会将消息声明为 Serializable：

```
public class ImmutableMessage implements Serializable {
    public final String string;
    public ImmutableMessage(String string) {
        this.string = string;
    }
}
case class ImmutableMessage(String: String)
```

理解不可变性不仅仅对于 Akka 消息是至关重要的，对于如何进行安全的通用并发编程也是必不可少的。因此，无论什么时候，只要需要在线程之间共享数据，就应该首先考虑将数据定义为不可变。既然我们已经知道了如何定义不可变消息，那么就可以开始学习如何使用各种消息模式将多个 Actor 组合起来了。

3.2.2 Ask 消息模式

Ask 模式会生成一个 Future，表示 Actor 返回的响应。Actor 系统外部的普通对象与 Actor 进行通信时经常会使用这种模式。我们之前已经介绍过 ask，知道它会返回一个表示响应的 Future，不过读者可能还不是很清楚，Akka 是如何知道用哪个消息来完成 Future 的呢？

在调用 ask 向 Actor 发起请求时，Akka 实际上会在 Actor 系统中创建一个临时 Actor。接收请求的 Actor 在返回响应时使用的 sender()引用就是这个临时 Actor。当一个 Actor 接收到 ask 请求发来的消息并返回响应时，这个临时 Actor 会使用返回的响应来完成 Future，如图 3-3 所示。

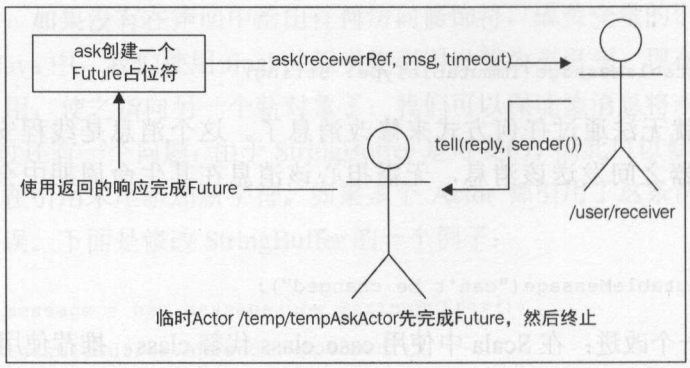


图 3-3

因为 sender()引用就指向临时 Actor 的路径，所以 Akka 知道要用哪个消息来完成 Future。知道这点之后，就可以将多个 Actor 组合起来，确保能够将响应发送到正确的位置，完成各种 ask 请求。

Ask 模式要求定义一个超时参数，如果对方没有在超时参数限定的时间内返回这个 ask 的响应，那么 Future 就会返回失败。ask/?方法要求提供的超时参数可以是长整型的毫秒数，也可以是 akka.util.Timeout，这种类型提供了更丰富的时间表达方式。

在 Java 中使用 ask 时，可以用 TimeUnit 来构造 Timeout，如下所示：

```
static import akka.pattern.Patterns.ask;
Timeout timeout = new akka.util.Timeout(
    1,
    java.util.concurrent.TimeUnit.SECONDS
);
Future future = ask(actor, message, timeout);
```

在 Scala 中，则可以使用 scala.concurrent.duration 来定义 Timeout。Scala 的 duration 领域特定语言（Domain Specific Languages，DSL）很强大，允许用户直接使用 1 second 这样的方式来定义一段时间。在 Scala 中，ask 的超时参数是隐式传入的，这样有助于简化 ask 的语义，并且使得 ask 语句更为简洁：

```
actorRef ? message
```

例如，导入了需要的库并且定义了隐式的 timeout 后，代码如下所示：

```
import scala.concurrent.duration._
import akka.pattern.ask
implicit val timeout = akka.util.Timeout(1 second)
val future = actorRef ? "message"
```

接着我们将介绍一个使用 ask 进行设计的例子，以此来展示如何将多个 ask 组合起来。由于 ask 会返回 Future，因此本质上是将 Actor 通过 ask 生成的 Future 组合起来。

使用 Ask 进行设计

首先，我们将展示如何使用 ask 模式来设计示例应用程序。这是最直观的方法。在这个例子中，文章解析服务将发起 ask 请求检查缓存，如果要解析的文章不在缓存中，就会向 HttpClientActor 发起 ask 请求，然后使用 ArticleParserActor 来解析得到结果。得到结果后，该服务会尝试将该文章存储到缓存中，最后将文章返回给用户，如图 3-4 所示：

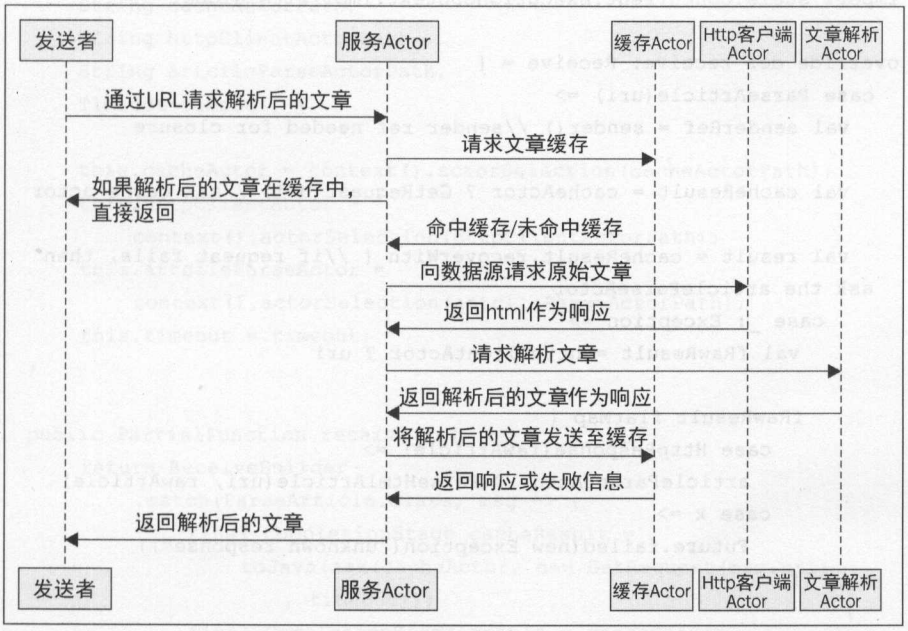


图 3-4

这是一个有助于读者理解的例子，不过并不是最佳的设计。如果使用同步 API，那么很可能就会选择这种设计。所以很适合以此作为起点并在此基础上进行优化。在继续介绍余下内容前，先来看一下相关的代码。

为了简单起见，我们将省略除了直接向客户端提供服务的 Actor 之外的其他 Actor 代码。读者可以在 GitHub 上本章的示例文件夹下找到完整的代码示例。在这个示例中，每个 Actor 要么返回一个错误，要么在成功情况下返回一个字符串表示的文章（原始文章或是解析过的文章）。

下面是 Scala 的源代码：

```
package com.akkademy.askdemo
```

```
class AskDemoArticleParser(
  cacheActorPath: String,
  httpClientActorPath: String, acticleParserActorPath: String,
  implicit val timeout: Timeout
) extends Actor {
  val cacheActor = context.actorSelection(cacheActorPath)
  val httpClientActor = context.actorSelection(httpClientActorPath)
  val articleParserActor = context.actorSelection(
    acticleParserActorpath)

  import scala.concurrent.ExecutionContext.Implicits.global

  override def receive: Receive = {
    case ParseArticle(uri) =>
      val senderRef = sender() //sender ref needed for closure

      val cacheResult = cacheActor ? GetRequest(uri) //ask cache actor

      val result = cacheResult.recoverWith { //if request fails, then
    ask the articleParseActor
        case _: Exception =>
          val fRawResult = httpClientActor ? uri

          fRawResult flatMap {
            case HttpResponse(rawArticle) =>
              articleParserActor ? ParseHtmlArticle(uri, rawArticle)
            case x =>
              Future.failed(new Exception("unknown response"))
          }
      }

      result onComplete { //could use Pipe (covered later)
        case scala.util.Success(x: String) =>
          println("cached result!")
          senderRef ! x //cached result
        case scala.util.Success(x: ArticleBody) =>
          cacheActor ! SetRequest(uri, x.body)
          senderRef ! x
        case scala.util.Failure(t) =>
          senderRef ! akka.actor.Status.Failure(t)
        case x =>
          println("unknown message! " + x)
      }
  }
}
```

下面是 Java 8 的代码：

```
public class AskDemoArticleParser extends AbstractActor {

    private final ActorSelection cacheActor;
    private final ActorSelection httpClientActor;
    private final ActorSelection artcileParseActor;
    private final Timeout timeout;

    public AskDemoArticleParser(
        String cacheActorPath,
        String httpClientActorPath,
        String artcileParseActorPath,
        Timeout timeout) {

        this.cacheActor = context().actorSelection(cacheActorPath);
        this.httpClientActor =
            context().actorSelection(httpClientActorPath);
        this.artcileParseActor =
            context().actorSelection(artcileParseActorPath);
        this.timeout = timeout;
    }

    public PartialFunction receive() {
        return ReceiveBuilder
            .match(ParseArticle.class, msg -> {
                final CompletionStage cacheResult =
                    toJava(ask(cacheActor, new GetRequest(msg.url)
                        , timeout));
                final CompletionStage result = cacheResult
                    .handle((x, t) -> { return (x != null)
                        ? CompletableFuture.completedFuture(x)
                        : toJava(ask(httpClientActor, msg.url,
                            timeout))
                        .thenCompose(rawArticle -> toJava(
                            ask(artcileParseActor,
                                new ParseHtmlArticle(msg.url,
                                    ((HttpResponse) rawArticle).body),
                                timeout))
                        );
                    }).thenCompose(x -> x);

                final ActorRef senderRef = sender();
                result.handle((x,t) -> {
```

```
                if(x != null) {
                    if(x instanceof ArticleBody) {
                        String body = ((ArticleBody) x).body;

                        //parsed article
                        cacheActor.tell(body, self()); //cache it
                        senderRef.tell(body, self()); //reply
                    } else if(x instanceof String) //cached article
                        senderRef.tell(x, self());
                } else if( x == null )
                    senderRef.tell(
                        new akka.actor.Status.Failure(
                            (Throwable)t),
                        self());
                return null;
            });
        }).build();
    }
}
```

Scala 和 Java 8 的例子功能完全相同：Actor 的构造函数有几个字符串参数，分别表示另外几个 Actor 的路径。跟之前一样，首先使用 actorSelection 查询得到这几个 Actor 的引用。我们使用依赖注入的方式访问这几个 Actor 路径，这样就可以对 Actor 路径进行配置。例如，用于缓存的 Actor 在测试环境下可能就在本机上，而在生产环境下可能就在一台远程机器上。

一旦接收到消息后，就可以尝试从缓存中获取文章。只要 cacheResult 没有成功返回，那么就在 recover/exceptionally 块中连续执行下面两个任务：

- 向作为 HTTP 客户端的 Actor 发送 ask 请求，请求获取原始文章；
- 向用于文章解析的 Actor 发送 ask 请求，请求对原始文章进行解析。

如果缓存请求失败，我们其实不需要关注造成失败的原因是什么。比如存储缓存的机器掉线了，此时仍然能够将结果返回给用户。不过可以把异常错误信息记录到日志，或是使用某种方法捕捉错误的原因。

最后，我们注册了一个回调函数，定义了得到结果或错误时的处理方法：如果成功，就把结果返回给用户；如果失败，就返回包含错误原因的失败信息。

使用 ask 是一种简单的解决方案，不过在 Actor 中选择 ask 作为主要的消息模式时，还是有一些注意要点。Ask 是一种用于构建简单解决方案的好方法，不过有时候使用 tell 来进行设计效果更好，稍后就会介绍 tell。接着先来看一下使用 ask 时需要注意的几个要素。

在另一个执行上下文中执行回调函数

在上面的例子中，注意到我们创建了一个本地的 ActorRef 变量，用于存储 sender() 方法的结果。这一点很重要：创建这个变量是必须的，当读者开始使用 Akka 编写代码后，至少要进行一次该操作。由于匿名函数是在一个不同的线程中执行的，有着不同的执行上下文，因此在匿名函数中的代码块里调用 sender()方法时，返回值是不可预知的。这个问题对于初学者来说并不明显。在老版本的 Scala Akka 例子中，sender()很多时候没有加上括号。现在的建议是，在 Scala 里永远给 sender()加上括号，原因在于 sender()方法并不是引用透明（Referential Transparent）的（多次调用的结果不一定会返回相同的结果），加上括号后可更明确地表示这是一个方法调用。为了访问正确的 ActorRef，就必须在主线程中调用 sender()，然后将结果引用存储在一个变量中。执行匿名函数时，这个变量会被正确地传递至匿名函数的闭包中。有一种更好的办法可以处理这个问题，叫做 Pipe，我们接下来将对此进行介绍。不过我们还是应该要理解闭包的这一重要特点。

必须设置超时参数

注意到示例中将同一个超时值传递给多个不同的 ask 请求作为参数。要向 Actor 请求获取响应，就必需创建一个明确的超时值。如果在超时值设定的时间范围内接收请求的 Actor 没有能够将响应返回至 Future，那么 Future 就会返回失败。

在缺乏运行着的生产系统提供的真实数据的情况下，要选择正确的超时值是很困难的。如果把超时值设置得太短，就可能会使得原本会成功的操作返回错误。如果设置得太长，又会强制用户在操作由于异常而失败时等待过长的时间。要合理地设置超时，就需要生产系统中的操作统计数据。由于我们无法控制依赖系统的性能，所以要设置正确相当困难。

由于每个 ask 请求都要求指定一个超时参数，所以当接收 ask 请求的 Actor 还要向另一个 Actor 发起 ask 请求时，还是使用同一个超时值就不太容易了。一旦某处发生错误，我们将在日志中找到好几个超时值，这会显著加大调试的难度，如图 3-5 所示。

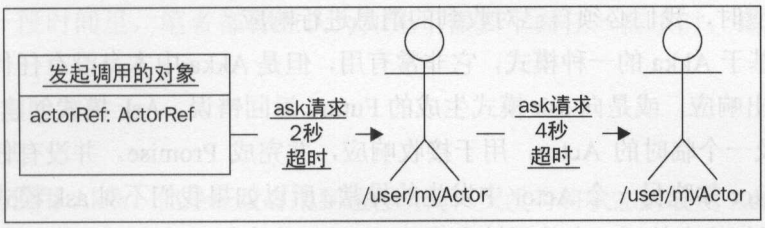


图 3-5

在图 3-5 中，即使所有代码都能够正确工作，所有系统都能够做出响应，2 秒的超时也有可能导致 Future 返回失败。

乍一看，似乎有个好办法：如果把超时参数设得很大，就可以避免发生类似的错误了。不过这样违反了我们在构建应用程序时想要遵守的响应式设计 4 准则中的灵敏性准则。所以可以认为，设置任意的或是很大的超时参数是一种反模式（Anti-pattern）。一个长达 30 秒的超时参数是基本没有实际意义的：等待数据的用户很可能在超时结束以前就放弃等待了。在大多数情况下，如果用户确实要等待超过 10 秒才能够完成某个操作，那么他们很可能再也不会使用这款软件了。当然也有一些特例，不过微软和谷歌都做过研究，结果显示：如果用户在使用 Web 应用程序时需要等待超过 2 秒才能看到页面，就会对他们的使用体验有负面影响。

超时错误的栈追踪信息并没有用

我们发起的每个 ask 请求都有超时参数。在本书例子中，整个操作包含了多个 ask 请求，所以有好几个地方可能发生超时错误。当 ask 请求超时的时候，异常会由 Akka 的调度器线程抛出，而不是由 Actor 的本地线程抛出。因此我们无法通过打印 AskTimeoutException 的栈追踪信息来判断出到底是哪个 ask 请求导致了超时。我们很难通过如下异常信息来调试应用程序：

```
akka.pattern.AskTimeoutException: Ask timed out on [Actor[akka://
system/user/actor#778114812]] after [2000 ms]
        at akka.pattern.PromiseActorRef$$anonfun$1.
apply$mcV$sp(AskSupport.scala:335)
        at akka.actor.Scheduler$$anon$7.run(Scheduler.scala:117)
```

另一点需要注意的是：如果 Actor 抛出了一个意料之外的异常，而没有返回错误，那么这个错误看上去会像是由于超时引起的，但是实际上却另有原因。

从这里总结出的经验就是：当代码中发生错误时，一定要返回失败消息。如果一个 Actor 抛出了异常，那么它是不会返回消息的。在 Actor 中，代码的编写者负责实现所有的消息处理逻辑：如果某个 Actor 需要进行响应，Akka 是不会隐式地做任何响应的。当需要返回响应时，我们必须自己对收到的消息进行响应。

Ask 是基于 Akka 的一种模式，它非常有用，但是 Akka 中本身没有任何机制可以自动对消息做出响应，或是向 ask 模式生成的 Future 返回错误。Ask 模式创建了一个 Scala Promise 以及一个临时的 Actor，用于接收响应，并完成 Promise。并没有任何机制能够使得临时 Actor 知晓另一个 Actor 中发生的异常，所以如果我们不对 ask 模式创建的临时 Actor 返回任何响应的话，它就无法完成 Promise，响应的 Future 就会发生超时错误，如图 3-6 所示。

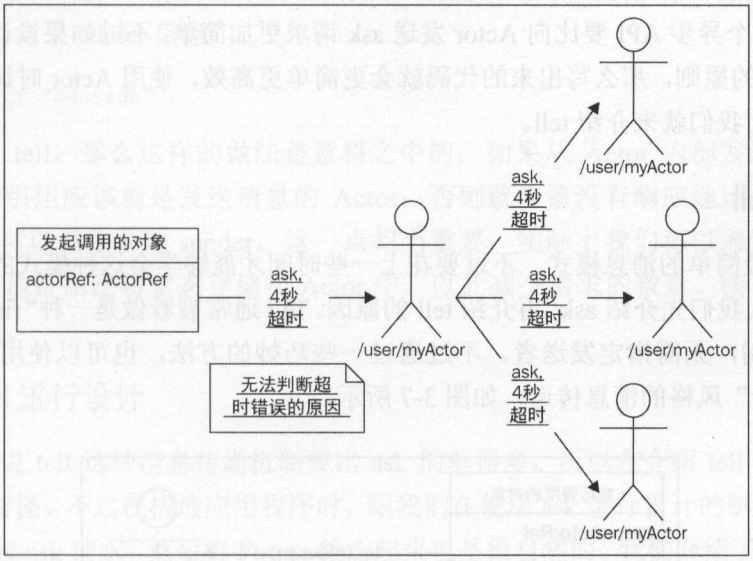


图 3-6

Ask 的额外性能开销

Ask 模式看上去很简单，不过它是有隐藏的额外性能开销的。首先，ask 会导致 Akka 在/temp 路径下新建一个临时 Actor。这个临时 Actor 会等待从接收 ask 消息的 Actor 返回的响应。其次，Future 也有额外的性能开销。Ask 会创建 Future，由临时 Actor 负责完成。这个开销并不大，但是如果需要非常高频地执行 ask 操作，那么还是要将这一开销考虑在内的。Ask 很简单，不过考虑到性能，使用 tell 是更高效的解决方案。

Actor 和 Ask 的复杂性

如果只是向 Actor 发送 ask 请求，而这些 Actor 并不包含状态，那么只要用 Future 基本就可以了。在前面的例子中，我们把 Actor 当做是一个异步 API 来使用，通过 ask 命令来调用。我们可以把 Actor 替换成返回 Future 的方法，这样代码是等价的，也更容易阅读。

在很长一段时间里，笔者都曾经认为如果不需要下面任一需求时，最好不要使用 Actor：

- 状态与并发；
- 分布式。

如果正在使用 Akka，但是既没有用到远程访问，又没有将状态封装在 Actor 中，支持并发状态访问，那么相较于无状态的非阻塞异步类，Akka 的优势可能就不那么明显了。

但是，笔者相信这是因为之前写的 Actor 代码设计比较糟糕。确实，如果没有状态的话，使用一个异步 API 要比向 Actor 发送 ask 请求更加简单。不过如果设计时采用“tell 优先于 ask”的原则，那么写出来的代码就会更简单更高效，使用 Actor 时调试起来也会更方便。下面我们就来介绍 tell。

3.2.3　Tell

Tell 是最简单的消息模式，不过要花上一些时间才能够学会这种模式的最佳实践。这也是为什么我们先介绍 ask，再介绍 tell 的原因。Tell 通常被看做是一种“fire and forget”消息传递机制，无需指定发送者。不过通过一些巧妙的方法，也可以使用 tell 来完成“request/reply”风格的消息传递，如图 3-7 所示。

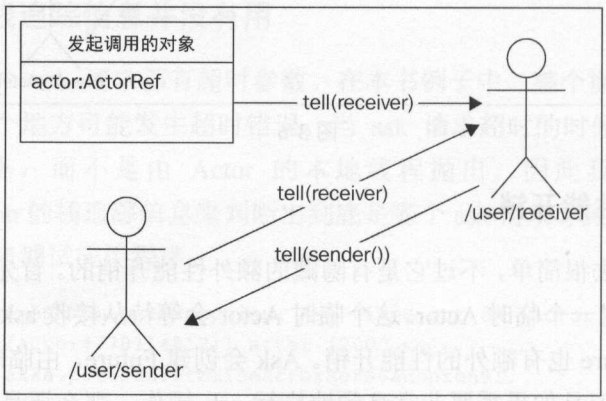


图 3-7

Tell 是 ActorRef/ActorSelection 类的一个方法。它也可以接受一个响应地址作为参数，接收消息的 Actor 中的 sender()其实就是这个响应地址。在 Scala 中，默认情况下，sender 会被隐式定义为发送消息的 Actor。如果没有 sender（如在 Actor 外部发起请求），那么响应地址不会默认设置为任何邮箱（叫做 DeadLetters）。

在 Java 中并没有隐式定义或是默认参数，所以必须提供 sender。如果不想指定任何特定的 sender 作为响应地址，那么应该遵循下述习惯。

- 从一个 Actor 内部发送消息，使用 self()：

```
actor.tell(message, self());
```

- 从 Actor 系统外部发送消息，使用 noSender：

```
actor.tell("message", akka.actor.ActorRef.noSender());
```

- 在 Scala 中 sender 信息是默认定义或隐式传入的，所以下面这种简单的语法已经隐式包含了 sender 信息：

```
actor ! "message"
```

如果使用 tell，那么这样的做法是意料之中的；如果从 Actor 内部发送消息，那么响应地址的引用应该就是发送消息的 Actor；否则就应该没有响应地址。不过，在使用 tell 时也可以指定其他 sender，这一点相当重要：实际上我们可以通过一些颇具创造性的响应地址用法将状态存储在 Actor 中，以此减少请求的数量。接下来我们将对此进行介绍。

使用 Tell 进行设计

理论上来说 tell 这种消息传递机制要比 ask 简单得多，所以在介绍 tell 之前先介绍 ask 似乎有点奇怪。不过在构造应用程序时，跟我们在使用 ask 进行设计的例子中那样向各个 Actor 发起 ask 请求，然后将 Future 组合起来也是很自然的。我们介绍了 ask 模式的几个问题以及超时和额外性能开销，这就给了我们动力去了解其他解决方案。通过使用 tell，我们可以做得更好。

Tell 通常被认为是一种 fire-and-forget 消息模式，不过在设计的时候，要获得更优秀的方案，就需要改变我们对于对象和 Actor 及其交互方式的看法。如果读者习惯使用 Scala 的话，可能会发现，如果在 Actor 中存储一些状态，并创建一些临时 Actor 来处理某些任务会使得设计变得更加简单。这和习惯函数式编程的程序员的直觉恰恰相反，他们通常会避免维护状态。面向对象语言设计之初也是消息驱动的，所以如果我们回顾一下 SmallTalk 语言中的一些优秀实践的话，也可以得到一些设计启发。

使用 Tell 处理响应

由于在返回消息时可以访问到指向发送者的引用，所以要对某条消息做出响应是很容易的。不过，在处理响应的时候，我们需要知道 Actor 收到的是哪一条消息的响应。如果我们在 Actor 中存储一些状态，记录 Actor 希望收到响应的消息，那么就能够高效地向 Actor 发送请求，解决前面提到的 ask 模式的问题。

下面举一个非常简单的使用 tell 进行设计的例子。我们可以在 Actor 中将一些上下文信息存储在一个 map 中，将 map 的 key 放在消息中一起发送。然后，当有着相同 key 的消息返回时，就可以恢复上下文，完成消息的处理了。这就允许我们能够使用类似 ask 模式的语义，又可以避免 ask 造成的额外性能开销，如图 3-8 所示。

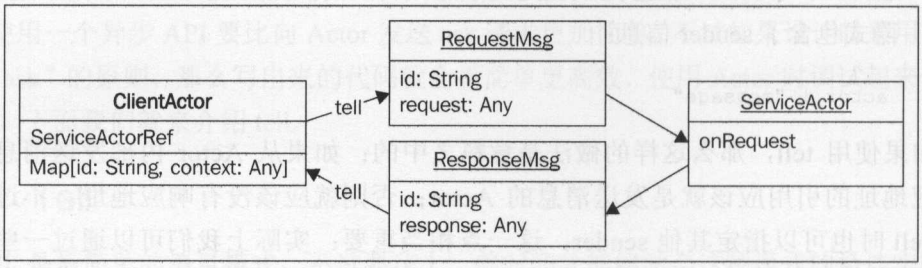


图 3-8

乍一看这种做法的性能开销似乎比使用 ask 时还要大，但是如果要把许多个 Actor 组合起来，这种做法就省去了使用 ask 时的超时以及创建额外 Actor 的问题。这样我们就能够控制超时发生的位置。

由于 tell 不需要提供超时参数，所以我们通常都会想要在某些时候及时生成超时信息。

对 Tell 进行超时调度

我们在这里将介绍 scheduler（这不是一个很重要的知识点）。scheduler 能够以一个固定的间歇重复发送消息；不过它最常见的用法是用于延迟 tell 的执行。例如，如果某个 Actor 想要在 3 秒中后向它自己发送一个用于“超时检查”的消息，scheduler 就是可以用来完成这一任务的机制。

在 Actor 中，我们经常希望在一段特定的时间段结束后发生某一事件，如果该事件没有发生，那么我们就可以用某种方式来表示失败。Scheduler 就可以用来调度这样的事件（比如超时）。下面的例子会在 3 秒钟之后向 Actor 发送一条“timeout”消息：

```
//Java
context().system().scheduler().scheduleOnce(Duration.create(3,
TimeUnit.SECONDS), actor, "timeout");

//Scala
context.system.scheduler.scheduleOnce(3 seconds, actor, "timeout")
```

我们马上就会看到这个例子的执行效果。

Tell 优先于 Ask（Tell Don’t Ask）：

“面向过程的代码先得到信息，然后做决定。而面向对象的代码则告诉对象要执行什么样的操作。”

——Alec Sharp《SmallTalk by Example》

设计过 SmallTalk 面向对象语言的设计者总结出了一条面向对象设计中非常重要的原则，使用 Actor 的编程范式来解释这个原则也相当合适：相较于 ask，优先使用 tell。

该原则的含义是：就和优秀的面向对象设计一样，消息应该表示命令，Actor 则应该是状态与行为的组合，而不应该由过程调用组成。在设计的时候，本书推荐在这条原则的基础上更进一步，避免在 Actor 之间使用 ask 模式，观察最后的设计会是什么样子。这只是一个启发，所以请自行判断设计的优劣，及其带来的开销以及益处。通常来说，我们会发现使用 tell 会产生更简单、更高效的消息流。

“Tell 优先于 Ask”其实更多的是一条面向对象设计的启发式原则，不过在介绍过 ask 模式的缺点之后，我们也可把这条准则应用到 Actor 的设计之中。Ask 是很简单的，所以在使用 Akka 时要判断一下是否应该使用 ask，但是要了解还有其他的解决方案。

避免在匿名 Actor 中使用 Ask

如果从 Actor 外部的普通对象中调用 tell，那么不使用 ask 的话并没有很明显的方法可以用来接收并处理响应。而我们之前介绍过，如果是在 Actor 之间进行交互，那么可以通过在 Actor 中捕捉关于当前消息的状态信息（例如 ID）来处理响应。除此之外，还有另一种方案可以用来处理响应：通过新建一个临时 Actor，我们可以描述对于某条消息的响应。

我们将在一个例子中使用 tell 来代替 ask。在这个例子里，我们将创建一个临时 Actor，用于处理消息的响应。注意到这和 Akka 处理 ask 的底层原理很类似：ask 也会创建一个临时 Actor，用于处理发送至 sender 的响应，并完成 Future。

我们将只展示 receive 代码块（先是 Java，然后 Scala）：

```
//Java
public PartialFunction receive() {
    return ReceiveBuilder
        .match(ParseArticle.class, msg -> {
            ActorRef extraActor = buildExtraActor(sender(), msg.url);
            cacheActor.tell(new GetRequest(msg.url), extraActor);
            httpClientActor.tell(msg.url, extraActor);
            context().system().scheduler().scheduleOnce(
                timeout.duration(),
                extraActor,
                "timeout",
                context().system().dispatcher(),
                ActorRef.noSender()
            );
        }).build();
    }

//Scala
```

```
override def receive: Receive = {
  case msg @ ParseArticle(uri) =>

    val extraActor = buildExtraActor(sender(), uri)

    cacheActor.tell(GetRequest(uri), extraActor)
    httpClientActor.tell("test", extraActor)

    context.system.scheduler.scheduleOnce(
      timeout.duration,
      extraActor,
      "timeout"
    )
}
```

由于这里没有生成任何 Future，所以代码看上去更简单。不过 buildExtraActor 方法创建了一个额外的 Actor。

下文中我们会展示 buildExtraActor 方法，不过在这之前先来解释一下上面的代码。receive 块得到 ParseArticle 消息后首先创建 extraActor。Extra 这个术语来自于 Jamie Allen 的书《Effective Akka》，由 O'Reilly 出版。在书中他展示了一种他称之为 Extra 模式的简单模式。

在创建了 extraActor 之后，receive 块继续发送三条消息：

- 向用于缓存的 Actor 发送一条消息，请求保存在缓存中的文章，用于缓存的 Actor 会将一个字符串返回至 sender。这里将 extraActor 作为 sender 参数传递给 tell。
- 向 httpClientActor 发送一条消息，请求原始文章。httpClientActor 会向 extraActor 返回一个 HttpResponse。
- 进行调度，向 extraActor 发送一条“timeout”消息。

这个例子与使用 ask 的例子有所不同，并不需要等待用于缓存的 Actor 返回响应。我们也可以在 extraActor 中发出这些请求，不过为了简单起见，我们在这里直接发送这几个消息，extraActor 先接收到哪条消息就对其进行响应。

extraActor 是一个匿名 Actor，定义了对上述 3 个消息的响应：

- 如果接收到来自缓存 Actor 的响应，就将该响应发送至 originalSender；
- 如果接收到一个 HttpResponse，就将它发送至 ArticleParser 进行解析，接收 ArticleParser 返回的 ArticleBody 作为响应；
- 如果接收到 ArticleBody，说明 HttpResponse 已经解析完成，所以将结果保存至缓存，然后将结果返回给 originalSender；
- 如果接收到“timeout”，就向 originalSender 返回失败。

extraActor 描述了对于 3 条消息的响应，包括如何获取以及如何解析原始 HTML 文章。这样就把所有这些行为放在了同一个匿名 Actor 中，而这个匿名 Actor 每次只能处理其中一种类型的消息，我们在每种情况下分别定义 3 条消息的响应，如图 3-9 所示。

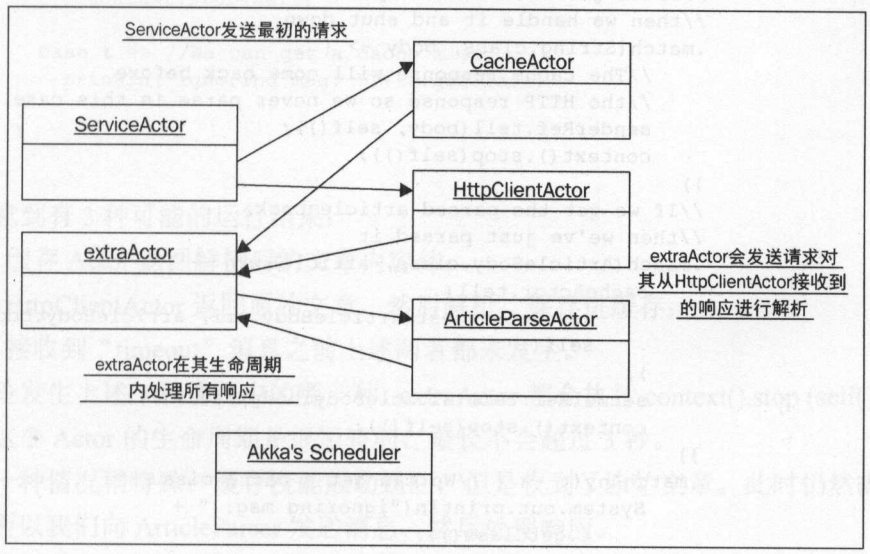


图 3-9

下面是用于构造 extraActor 的工厂方法：

```
//Java
private ActorRef buildExtraActor(ActorRef senderRef, String uri) {

    class MyActor extends AbstractActor {
        public MyActor() {
            receive(ReceiveBuilder
                .matchEquals(String.class, x ->
                    //if we get timeout, then fail
                    x.equals("timeout"), x -> {
                        senderRef.tell(
                            new Status.Failure(
                                new TimeoutException("timeout!")
                            ),
                            self()
                        );
                        context().stop(self());
                    })
                .match(HttpResponse.class, httpResponse -> {
                    //If we get the cache response first,
                    //then we handle it and shut down.
                    //The cache response will come back before
                    //the HTTP response so we never parse in this case.
```

```
                    artcileParseActor.tell(
                        new ParseHtmlArticle(uri, httpResponse.body),
                        self()
                    );
                })
                //If we get the cache response first,
                //then we handle it and shut down.
                .match(String.class, body -> {
                    //The cache response will come back before
                    //the HTTP response so we never parse in this case.
                    senderRef.tell(body, self());
                    context().stop(self());
                })
                //If we get the parsed article back,
                //then we've just parsed it
                .match(ArticleBody.class, articleBody -> {
                    cacheActor.tell(
                        new SetRequest(articleBody.uri, articleBody.body),
                        self()
                    );
                    senderRef.tell(articleBody.body, self());
                    context().stop(self());
                })
                .matchAny(t -> { //We can get a cache miss
                    System.out.println("ignoring msg: " +
                        t.getClass());
                } }
                .build());
        }
    }
    return context().actorOf(Props.create(
        MyActor.class, () -> new MyActor()
    ));
}

//Scala
private def buildExtraActor(senderRef: ActorRef, uri: String):
ActorRef = {
  return context.actorOf(Props(new Actor {
    override def receive = {
      case "timeout" => //if we get timeout, then fail
        senderRef ! Failure(new TimeoutException("timeout!"))
        context.stop(self)
      case HttpResponse(body) => //If we get the http response
      first, we pass it to be parsed.
        articleParserActor ! ParseHtmlArticle(uri, body)

      case body: String => //If we get the cache response first,
      then we handle it and shut down.
        //The cache response will come back before the HTTP response
        so we never parse in this case.
        senderRef ! body
        context.stop(self)
```

```
        case ArticleBody(uri, body) => //If we get the parsed article
        back, then we've just parsed it
          cacheActor ! SetRequest(uri, body) //Cache it as we just
          parsed it
            senderRef ! body
            context.stop(self)

        case t => //We can get a cache miss
          println("ignoring msg: " + t.getClass)
      }
    }))
  }
```

注意到有 3 种可能的运行结果：

- 缓存 Actor 返回解析后的文章内容体；
- HttpClientActor 返回原始文章，然后解析，保存进缓存；
- 接收到“timeout”消息之前上述两者都未发生。

无论发生上述 3 种情况中的哪一种，extraActor 都会执行 context().stop (self())来终止运行。这个 Actor 的生命周期是很短暂的，最长不会超过 3 秒。

有一种情况稍特殊：缓存没能成功返回，但是收到了原始文章。此时仍然需要解析文章，所以我们向 ArticleParser 发送消息，然后处理响应。

这个例子看上去代码反而更多了，但实际上它比使用 ask 的例子更加便捷：

- 使用 ask 的例子会为每个 ask 请求都创建一个 Future；
- 使用 ask 的例子会为每个 ask 请求都创建一个临时 Actor。

我们的这个例子没有创建任何 Future，只创建了一个额外的 Actor。而且分析错误的时候也更加简单：

- 在使用 ask 的例子中，有 3 个不同的超时可能会导致错误。

而在这个例子中，只有一个我们可以控制的超时：要么发生超时，要么运行成功。在这个方法中，我们还可以在发生超时时将临时 Actor 中的状态输出到日志，这有助于理解到底出了什么问题。相较而言，Future 的超时信息就不是很有用了。

使用 tell 只需要比使用 ask 的例子多写一点代码，就可以自己控制 Actor 处理响应的行为，而不需要依赖 Akka 的 ask 模式的实现细节。好处就是我们能够根据具体的用例来构建合适的解决方案。笔者也曾经在一些更喜欢简单紧凑的代码的团队中工作过，所以到底选择 tell 还是 ask 由开发者及其团队来决定。请探究理解这两种方法的优缺点。对于需要高性能的代码来说，tell 更为高效。

这里有个很有效的练习：修改 extraActor 的代码，使之只在缓存请求无法返回结果时才请求原始文章并请求对其进行解析。不要害怕在 extraActor 中保存状态，这样有助于理解它处在生命周期中的哪个状态。

这个例子展示了如何对一个使用多个 ask 的设计进行修改，通过使用一个额外的 Actor 来处理响应并控制 Actor 之间的消息流来彻底避免使用 ask。并不存在万能的方案，不过我们应该尝试对不同的设计进行评估，选择最适用于具体用例的方案。

Forward

Tell 在语义上是用于将一条消息发送至另一个 Actor，并将响应地址设置为当前的 Actor。而 Forward 和邮件转发非常类似：初始发送者保持不变，只不过新增了一个收件人。

在使用 tell 时，我们指定了一个响应地址，或是将响应地址隐式设为发送消息的 Actor。而使用 forward 传递消息时，响应地址就是原始消息的发送者，如图 3-10 所示。

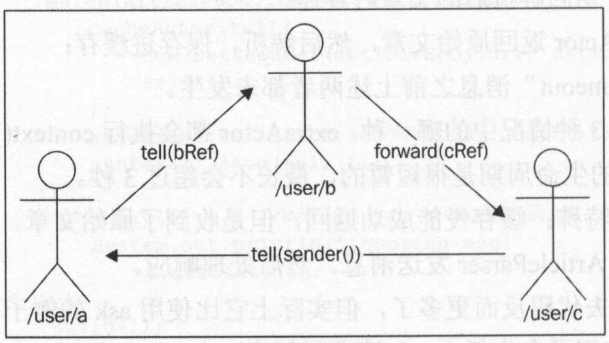


图 3-10

有时候我们需要将接受到的消息传递给另一个 Actor 来处理，而最终的处理结果需要传回给初始发起请求的一方。此时 forward 是很有用的。

处理中间步骤的 Actor 转发接收到的消息，或是发送一条新消息，但是仍然会将初始发送者与新消息一起发送。从银行账号中获取历史就是一个使用 forward 的例子。可能有一个 HistoryActor 负责接收查询账户历史的请求，然后将请求转发给 ChequingAccountActor 或是 CreditCardAccountActor。请求账户历史的消息会被转发给更具体的 Actor 来处理。

Forward 没有任何特殊之处，所以我们可以用 ActorRef.tell（message，sender）并指定 sender 来完成一模一样的任务。不过 forward 的语义更加清晰（尤其是在 Scala 中，很多地方都会使用!方法）。

```
//Java
actor.forward(result, getContext());

//Scala
actor forward message
```

要注意的是，在 Scala 中，context 信息是隐式传入的。

Pipe

很多情况下，需要将 Actor 中的某个 Future 返回给请求发送者。上文介绍过 sender()是一个方法，所以要在 Future 的回调函数中访问 sender()，我们必须存储一个指向 sender()的引用：

```
//Java
final ActorRef senderRef = sender();
future.map(x -> {senderRef.tell(x, ActorRef.noSender())});

//Scala
val senderRef = sender();
future.map(x => senderRef ! x);①
```

注册的 Future 回调函数（比如这个例子中的 map）会在另一个线程中执行，所以未必能够通过 sender()访问到正确的值。将 sender()存储起来的原因并不是那么清楚，这样的代码也不是很容易理解。对于没有自己遇到过这个问题的人来说，这么写似乎并没有什么道理。当读者读到这里的时候，也很可能会觉得为什么不直接在匿名函数中调用 sender()呢？请自己进行实验，就会发现在匿名函数中调用 sender()时，会返回出乎意料的不同结果。

而通过使用 Pipe 模式，就可以避免产生所有这些疑惑之处了。Pipe 会得到一个 Future 的结果，然后将 Future 的结果返回给 sender()，无论是成功结果，还是失败结果，都是我们最终希望得到的。

```
//Java
pipe(future, system.dispatcher()).to(sender());

//Scala
future pipeTo sender()
pipe(future) to sender()
```

pipe 接受 Future 的结果作为参数，然后将其传递给所提供的 Actor 引用。在上面的例子中，因为 sender()执行在当前线程上，所以我们可以直接调用 sender()，而不用干一些奇怪的事情（比如把 sender()引用存储在一个变量中），执行结果一切正确。这就好多了！

上述问题在相关工作中很容易发生。一旦开始在日志中发现 dead-letter 消息以后，就可以问问自己到底出了什么问题。不过只要碰到过一次这个问题，就可以拍拍胸脯认

① 译者注：原文为 future.map(x => senderRef ! ActorRef.noSender)，有误。

为自己开始有点儿懂异步编程了。在使用 Akka API 的过程中（尤其是使用 ask 的时候），一定会学到这宝贵的一课。

3.3　课后作业

在此回顾一下本章内容。

基本知识

在查看 GitHub 上提供的源代码之前，试着自己编写一下本章中的例子。

运用 ask 的设计，编写一些 Actor 来完成一些任务，并存储结果的缓存。

- 使用 pipe 来优化设计。
- 使用“Tell 优先于 Ask”原则，在不使用 ask 的条件下完成相同的工作。
- 试着使用“Tell 优先于 Ask”原则，利用 extraActor 将多个 Actor 之间的请求组合起来。

有可能会使用到一个匿名 Actor：

- 在上面的设计中，调度执行一个超时操作；
- 将用于缓存的 Actor 设置为远程 Actor，并完成上面的工作。

项目作业

现在我们已经能够处理许多类型的数据流了。

选择一个功能，试着用 ask 来实现：

- 试着进行修改，避免使用 ask。
- 新的设计到底好不好？可访问网上论坛并参与讨论！

3.4　小结

本章介绍了 Akka 中比较基本的消息概念，不过读者应该能够从中得到启发，慢慢成长为优秀的设计师。

现在我们应该已经对下面的消息模式有了基本的理解：

- Ask；
- Tell；
- Forward；
- Pipe。

本章还介绍了一些更高级的内容，比如 ask 的工作原理、使用 ask 时可能会遇到的问题、如何通过 Pipe 来简化 Future 的使用以及如何使用 Forward 来简化消息的转发等。在下一章中，我们将学习在 Actor 中处理状态的不同方法。

第 4 章
Actor 的生命周期——处理状态与错误

本章将介绍 Actor 的生命周期。当 Actor 遇到异常状态时会发生什么，如何通过修改 Actor 的状态来改变它的行为？在介绍响应式设计准则时我们提到过容错性，也介绍过如何在 Actor 中存储状态。本章将对这些话题展开讨论。

在开始介绍错误和状态之前，首先介绍所谓的分布式计算误区（The Fallacies of Distributed Computing）：开发者对于在网络之间通信的系统的一些常见误解。

本章将介绍下述话题：

- 分布式计算的误区；
- 当 Actor 运行失败时会发生什么；
- 如何通过监督 Actor 来处理失败；
- 如何利用 become()以及有限自动机来修改 Actor 的行为。

4.1 分布式计算的 8 个误区

在介绍例子之前，我们先快速介绍一下分布式计算的误区。分布式计算误区由 Sun Microsystems 的一个团队总结出来，包含缺乏经验的开发者对于在网络上通信的系统的一些错误的假设。接下来具体介绍一下这 8 个误区。

4.1.1 网络是可靠的

“网络是可靠的。”这是我们最多听到的一个误区，也可能是对如今的系统影响最大的一个误区。其实网络是不可靠的。我们很容易会用和处理本地系统相同的方式来处理远程系统，也就会像跟本地 Actor 进行交互一样和远程 Actor 进行交互。而 Akka 在这个

错误的假设上更进一步，通过提高网络通信的抽象层次为我们提供了位置透明性。

我们知道服务可能正在运行，也可能停止运行。可能到底满足什么条件才能说明服务是真正可用的呢？即使某个服务正在运行，也必需提供相应的网络传输才能访问该服务。如果在一个小型网络内有两台服务器，很容易就会认为这两台服务器是互相连接的，并且永远能够成功通信。然而它们两者之间最终总是可能会有地方出问题。比如路由器可能会运行崩溃并且重启，可能会停电，有人可能会拔错了数据中心的某根线缆等等。当我们开始扩展系统时，比如说有 1000 台或是好几千台服务器在运行，那么服务变得不可用的可能性会越来越高。在 Amazon Web Serve（AWS）上，发生暂时的网络错误是极为普遍的情况。我们现在部署在 AWS 上的系统每天都有发生连接失败和网络错误的时候。

4.1.2 没有延迟

在网络上传输消息是需要时间的，返回响应同样需要时间。而相较于运行在本地内存中的 Actor，与远程 Actor 通信会导致更长的延迟。真正在网络上开始传输任何数据之前，其实还发生了一些事情。

主机名会被解析成 IP 地址（通过 DNS），IP 地址会被解析成接收方的 MAC 地址（通过 ARP），并通过握手建立 TCP 连接，然后才开始传输数据。通过 TCP 传输的数据以连续的包发送，会接收到返回的认证包。这一过程是相当复杂的，可以想象，向谷歌发起一个搜索请求时，仅仅是这个请求离开本地机器到达谷歌的过程就要花上一些时间。

由于网络通信的额外开销，我们可以做出假设：相较于发送很多包含小型消息的请求，更好的做法是减少请求的数量，增大每个请求包含消息的大小。要把这一点实际应用到我们的示例数据库中，就可能需要添加一个在单个消息中发送多个 SetRequest 或 GetRequest 的功能。

要记住，在开发时使用本地机器内存中的对象来进行测试时没有问题的任务，一定要考虑网络引入的延迟，以及网络延迟对用户体验的影响。这就又回到了我们提到过的一个响应式准则：灵敏性。灵敏性不仅仅对于实时系统至关重要，Web 应用程序的灵敏性和第一人称射击游戏以及股票交易系统的灵敏性一样重要。

4.1.3 带宽是无限的

以如今的网络技术，带宽一直在稳定地增长着，所以对这个问题不需要过多地考虑。但是我们仍然不希望在网络上发送多余的数据。网络也是有容量的，必须要把这一点考虑进去。网络上传输的东西越多，通信被噪声数据影响的可能性就越大。同样地，在网络上传输的数据越多，花的时间也就越长。

对于延迟，我们讨论了如何在数据库中支持用单个消息处理多个 SetRequest 和 GetRequest，以减少网络上请求的数量。而对于带宽，我们则要努力减少网络上消息的大小。在 HTTP 通信中，经常会使用 Gzip 来减少每个请求中包含的数据量。在大型的分布式系统中，经常会使用能够在 CPU 上快速执行的压缩算法，比如 Snappy 和 LZO（例如，Cassandra 将 LZO 用于节点内部的通信）。

如果要将对这一点的考虑实际应用到数据库中，就要通过压缩来减少消息的大小。压缩也是有开销的：它会消耗 CPU 资源。有一些专用的算法把重点放在压缩效率，而非压缩比上。我们可以使用其中之一（比如 Snappy），在将序列化后的消息发送到远程 Actor 前先对其进行压缩，接收到消息后先解压缩，再反序列化。

4.1.4　网络是安全的

有一点很容易会被忽视：我们使用的网络并不是安全的。

在公共网络中（如一个共享的 Wi-Fi 网络），任何 HTTP 网络数据都可能会被拦截或窃听，而且这并不需要太多的技术知识。MySpace 的登录页面以前在 HTTP 上传输时是不加密的，所以人们坐在咖啡店里使用 Wi-Fi 很容易被盗取隐私信息。即使使用 HTTPS，还是有可能被中间人攻击通过 ARP 哄骗拦截并窃听，用户还是会得到一个证书错误，人类是系统中最弱的一环。

要记住的是：其他人永远都可以读取我们在网络上发送的数据，所以需要确保使用安全加密算法（如 AES-256）对传输的敏感信息进行保护，并且小心地保存处理密钥。

要把这一点应用到数据库，就要支持对通信进行加密，并且验证发送者的身份。

4.1.5　网络拓扑不会改变

现在，当我们构建了一个应用程序，并且将其部署到公司的基础设施上或是部署到云端时，虽然 IP 地址看上去是类似的，但是已经不是由我们自己控制的了。如果系统依赖于特定的 IP 地址，那么就做出了错误的假设：网络不会发生变化。部署到云端时，这个问题就更加严重了：每次停止并重启机器都会得到一个新的 IP。

我们可以使用 DNS 或者 Zookeeper 等技术提供的服务注册来减少这个问题带来的影响，不过重要的是：要意识到，网络是会发生变化的。

4.1.6　只有一个管理员

在笔者曾经干过的几份工作中，曾有权限在某些情况下访问几个主要产品的生产系统（通常是需要在生产系统中进行工作的时候）。而作为开发者，我们可能不会意识到有

多少人会登入或登出我们的网络和机器。我们可能会与某个外部的数据中心或是云基础设施进行交互，这时候要记住：有人可能会离开公司，而不同的规则也可能会要求多个人有访问系统的权限等。

我们为什么要关心这一点呢？有一天我在测试环境部署了一个软件，并且启动了应用程序，然后马上发现了程序有问题。我没有意识到有人已经登录了系统，并且修改了一些配置。在管理软件时，我们应该尝试尽可能减少能够管理软件的入口（尤其是关于配置的入口）。不是每个人都有工具或是能够理解可能发生的改变会带来的影响。对于某个组件的一个小小的修改可能会对其他系统带来不可预料的影响。

4.1.7 网络传输没有开销

将数据向下传递到传输层也是有开销的。而数据序列化、数据压缩、将数据压入 TCP 缓冲区以及建立连接都要消耗资源。网络上的任何操作都是有开销的。

无论是在互联网还是云基础设施中，对网络的使用都是要花费真金白银的。基础设施的维护和云服务的使用都是有开销的。

4.1.8 网络是同构的

在脑海中想象一个网络。有数据从互联网通过负载均衡器转发进入防火墙内。该数据接着会被转发到运行在虚拟机（VM）上的应用程序内。现在假设我们构建了相同的应用程序，但是要部署到另一个环境中。

另一个网络环境是一样的吗？很有可能每个组件都是不同的。防火墙有可能是 Cisco 的，也可能是一个家庭路由器。负载均衡器可能是 F5 的，也可能是运行在 VM 上的 nginx 实例。我们的应用程序有可能运行在 IBM P595 上，也有可能运行在一个 VMWare ESXi 主机上。VM 可能运行的是 Ubuntu 操作系统，也可能运行 IBM AIX Unix 操作系统。

负载均衡器使用的是 round-robin 算法还是 sticky-sessions 算法呢？甚至说它支持不支持 sticky-sessions 算法呢？这些都是我们需要考虑的重要问题。我见到过一个部署到生产环境的应用程序使用了一个早期云提供商提供的负载均衡器，这个负载均衡器就不支持 sticky-sessions 算法。有时候一个有状态的应用程序被部署到了无状态的冗余备份上，由于我们假设网络是同构的，又假设所有的负载均衡器都是一样的，所以这一情况是意料之外的。然而其实并非如此。

4.2 错误

关于分布式系统的第一个误区就是认为网络是可靠的。其实网络并不可靠。网络是

会发生错误的。所以我们编写的所有代码都要正确地考虑可能发生的错误：网络会发生故障，消息会丢失。

还记得响应式宣言吗？其中的一个信条就是要能够灵活地应对错误。也就是说，必须要假设可能会发生错误情况。使用 Akka 的一个附带的好处就是容错性。在本小节中，我们将介绍 Akka 如何帮助我们来处理失败情况。

作为本小节中的例子，我们将了解在数据库中应对失败情况的不同方法。

4.2.1　隔离错误

在学习 Akka 如何对失败情况进行响应之前，先了解一些在分布式应用程序中都应该遵循的通用策略：隔离错误。

假设每个组件都是一个定时炸弹，那么我们希望能够确保无论其中任何一个发生爆炸，都不会引发链式反应，导致其他组件也爆炸。也可以说，我们希望能够隔离错误，或是将可能引发失败情况的组件分离开来。

船舱的隔板是一个很好的类比。笔者在 Jamie Allen 和 Roland Khum 的《响应式设计模式》（Reactive Design Patterns）一书中第一次看见使用这个例子来做类比。船和飞机通常各自都使用单独的船体来进行隔离，每个船体都有独立而又垂直的墙。如果某个船体破裂了，只有一个部分会进水，所以就算是船撞到了岩石或是冰川，由于其他部分都没有进水，因此船还是能够继续浮在水面。这是很好的例子说明了应该如何设计系统，使之能够应对失败情况。

冗余

保持系统在发生错误时仍能运行的方法之一就是实现各组件的冗余性，确保不存在单点故障。假设我们有一个服务，那么有多种方法可以通过冗余设计来保证服务的高可用性。

下面是使用 broker 的一个例子。broker 通常用于在不停止系统的前提下添加新的服务节点或是关闭某个服务节点。服务会连接到 broker，从消息队列中获取并处理消息。JMS 和 RabbitMQ 就是 broker 的例子。数据库可以用于存储 broker 的消息。注意到单个的 broker 本身就会导致单点故障，也可能会成为性能瓶颈，如图 4-1 所示。本书将介绍不使用 broker 的服务集群。

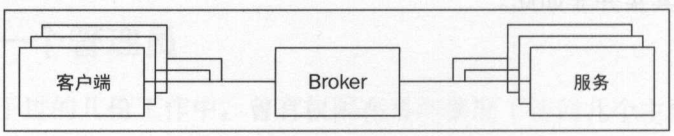


图 4-1

4.2.2 监督

Erlang 将容错性引入了 Actor 模型，它使用的概念叫做监督（supervision）。监督的核心思想就是把对于失败的响应和可能引起失败的组件分隔开，并且把可能发生错误的组件通过层级结构来组织，以便管理。

监督的层级结构

Akka 使用 Actor 层级结构来描述监督。当我们创建 Actor 时，新建的 Actor 都是作为另一个 Actor 的子 Actor，父 Actor 负责监督子 Actor。Actor 的路径结构就展示了它的层级结构，所以和文件系统中的文件夹有点像。

位于 Actor 层级结构顶部的是路径为/的根 Actor。然后是路径为/user 的守护 Actor。使用 actorSystem.actorOf() 函数创建的所有 Actor 都是守护 Actor 的子 Actor（/user/yourActor）。

如果在一个 Actor 内部创建另一个 Actor，那么可以通过调用 context().actorOf 使得新建的 Actor 成为原 Actor 的子 Actor。这个新建的 Actor 就成为了 Actor 树结构中的下一个叶节点（/user/yourActor/newChild）。

根 Actor 下面还有其他 Actor 层级结构。系统操作相关的 Actor 在路径为/system 的系统守护 Actor 下面。路径/temp 下面还包含了一个临时的 Actor 层级结构，用于完成 Future 的临时 Actor 就处于这个子树中。我们并不需要对这些细节担心过多，这些基本上都是 Akka 的内部实现，对于开发者是不可见的。

还记得寿司餐厅的例子码？假设这个餐厅老板负责管理一个经理，而经理直接管理两个员工（寿司师和服务员）。那么我们就可能会有如下所示的层级结构，如图 4-2 所示。

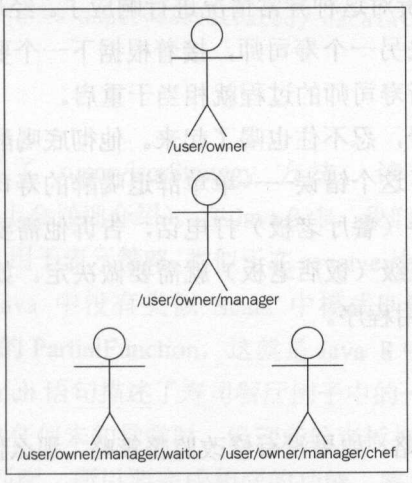


图 4-2

每个负责监督其他 Actor 的 Actor 都可以自行定义监督策略。下面我们就来介绍监督策略。

监督策略和醉酒的寿司师

我们将通过一个例子来帮助理解监督策略。假设寿司师很喜欢喝酒。他喝了酒以后经常给自己惹麻烦，所以此时他的经理必需要采取一些措施。经理可以选择一些不同的做法。下面我们逐一介绍。

- **继续**（resume）：Actor 继续处理下一条消息；
- **停止**（stop）：停止 Actor，不再做任何操作；
- **重启**（restart）：新建一个 Actor，代替原来的 Actor；
- **向上反映**（escalate）：将异常信息传递给下一个监督者。

为了帮助我们理解上面的概念，假设表示寿司师的 Actor 必需要制作寿司。他非常熟练，每做完一盘寿司，都要喝一杯酒来庆祝一下自己的杰作。服务员把客人点的菜单叠起来交给寿司师，寿司师就不断地制作寿司并喝酒。如果寿司师从早忙到晚的话，最后就无法处在最佳工作状态了（说得轻一点）。

如果寿司师犯了一个错误（比如说切到了自己的手指或是掉了一个盘子），那么这可能是可以接受的。监督者将告诉寿司师在这种情况下 resume()。

如果寿司师累了，犯下了错误，这说明他可能需要休息一下。这时候经理就可能让寿司师 Actor 停止处理消息，休息一下。通常都由监督者来决定在这种情况下采取哪种必要的行为，比如说，监督者也需要告诉他监督的 Actor 何时重新继续开始工作。

一旦寿司师喝醉了，做了一盘巨丑无比的寿司，开始让一些客人不满了，那么经理（寿司师的监督者）就要负责对这种异常情况进行响应了。经理要说明寿司师喝醉了，并且把他辞退了。监督者请来另一个寿司师，接着根据下一个要做的菜单制作寿司。送走原来的寿司师，引进一个新寿司师的过程就相当于重启。

新来的寿司师比较年轻，忍不住也喝了起来。他彻底喝醉了，还打翻了一支蜡烛，餐厅着火了。经理处理不了这个错误——单单辞退喝醉的寿司师，再新请一个师傅可灭不了火！经理给他的监督者（餐厅老板）打电话，告诉他需要关门并且马上报警。这就是向上反映，此时经理的上级（饭店老板）就需要做决定。如果把异常向上反映给了守护 Actor，那么就会关闭应用程序。

定义监督策略

Actor 有默认的监督策略。如果没有修改监督策略，那么监督 Actor 的行为基本上和喝醉的寿司师例子中一样：

- Actor 运行过程中抛出异常：restart()；
- Actor 运行过程中发生错误：escalate()；
- Actor 初始化过程中发生异常：stop()。

在默认监督策略中还定义了另一种情况：ActorKilledException。如果 Actor 被“杀”（kill）了，那么这个 Actor 的监督者会接收到一个 ActorKilledException，执行 stop()会接收到该异常。

还记得/system 这个路径吗？所有的监督事件实际上都发生在这个路径下面的 Actor 中，而且异常事件并不是普通的消息。如果想要试着推导消息传递的顺序，那么理解这一点是十分重要的。

我们来看一下如果想要自己描述经理的监督行为，应该如何定义监督策略。

Java 代码：

下面的 Java 代码展示了经理的监督策略：

```
@Override
public akka.actor.SupervisorStrategy supervisorStrategy() {
    return new OneForOneStrategy(5, Duration.create("1 minute"),
        akka.japi.pf.DeciderBuilder
            .match(BrokenPlateException.class,
                e -> SupervisorStrategy.resume())
            .match(DrunkenFoolException.class,
                e -> SupervisorStrategy.restart())
            .match(RestaurantFireError.class,
                e -> SupervisorStrategy.escalate())
            .match(TiredChefException.class,
                e -> SupervisorStrategy.stop())
            .matchAny(e -> SupervisorStrategy.escalate())
            .build()
    );
}
```

我们在 Actor 中重写了 supervisorStrategy 方法，该方法返回一个新的策略（OneForOneStrategy，我们马上会详细介绍）。在 Java 8 中，我们使用 DeciderBuilder 来创建一个 Scala 的 PartialFunction，用于表示策略，类似于在 reveive 块中用来构造 PartialFunction 的 ReceiverBuilder。虽然 Java 中没有类似 Scala 中模式匹配的语义，但是可以通过 DeciderBuilder 来构造 Scala 的 PartialFunction，这就是 Java 8 中用于模式匹配的 API，也相当好。我们分别用一个 match 语句描述了寿司餐厅例子中的一种场景，最后还加入了一个 matchAny 语句，表示收到任何未知异常时，经理就给老板打电话。

由于 Scala 提供了模式匹配，所以要完成相同的功能，需要写的模板代码更少：

```
override def supervisorStrategy = {
  OneForOneStrategy() {
    case BrokenPlateException => Resume
    case DrunkenFoolException => Restart
    case RestaurantFireError =>Escalate
    case TiredChefException => Stop
    case _ => Escalate
  }
}
```

在 Scala 的例子中，我们还是重写 Actor 的 supervisorStrategy 方法，然后定义一个 PartialFunction，匹配抛出的异常，并返回 Directive。在这里我们也针对例子中的所有场景定义了处理方法，并且在发生任何未知情况的时候向上反映。

无论是在 Scala 还是在 Java 中，我们都描述了当 Actor 抛出不同的 Throwable 时该采取什么样的处理方法：如果寿司师打破了盘子，就让他继续使用下一个盘子；如果寿司师喝醉了，就辞退他，再请一个新的寿司师；如果寿司师把餐厅弄着火了，就要让经理打电话给他老板；如果寿司师累了，就让他休息一下。

要注意的是，一般情况下使用默认的行为就可以了：如果 Actor 在运行中抛出异常，就重启 Actor；如果发生错误，就向上反映或是关闭应用程序。不过如果 Actor 在构造函数中抛出异常，那么会导致 ActorInitializationException，并最终导致 Actor 停止运行。因为在这种情况下应用程序不会继续运行，所以要对此特别注意。

Actor 生命周期

我们介绍了重启的概念，就好像把一个工人辞退，再雇一个新工人来代替他。现在我们就来学习 Actor 的生命周期中都发生了什么。

在 Actor 的生命周期中会调用几个方法，我们在需要时可以重写这些方法。

- prestart()：在构造函数之后调用。
- postStop()：在重启之前调用。
- preRestart(reason, message)：默认情况下会调用 postStop()。
- postRestart()：默认情况下会调用 preStart()。

在 Actor 生命周期中个事件的发生顺序如图 4-3 所示。

这一点对于设计是很有帮助的。假设有一个聊天应用程序，每个用户都用一个 Actor 来表示，而 Actor 之间的消息传递就表示了用户之间的聊天。当一个用户进入聊天室的时候，就启动了一个 Actor，并发送一条消息，更新聊天室中的用户名单。当一个 Actor 停止的时候，发送另一条消息，将该用户从聊天室中显示的当前活跃名单中删除。如果

使用默认实现，那么要是 Actor 遇到了异常并且重启，那么就会先将该用户从聊天窗口中删除，再将其添加进去。我们可以重写 preRestart 和 postRestart 方法，这样用户就只会在真正加入或离开聊天室的时候才被添加或删除到名单中，如图 4-4 所示。

注意

要注意的是preRestart和postRestart只在重启的时候才会被调用。它们默认调用了 preStart 和 postStop，但是调用它们的时候就不再直接调用 preStart 和 postStop 了。这样我们就能够决定，到底是只在 Actor 启动或停止的时候调用一次 preStart 和 postStop，还是每次重启一个 Actor 的时候就调用 preStart 和 postStop。

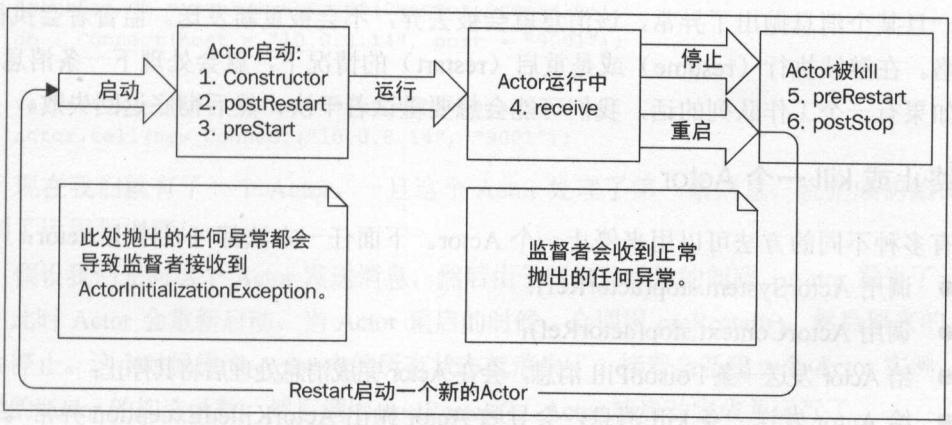


图 4-3

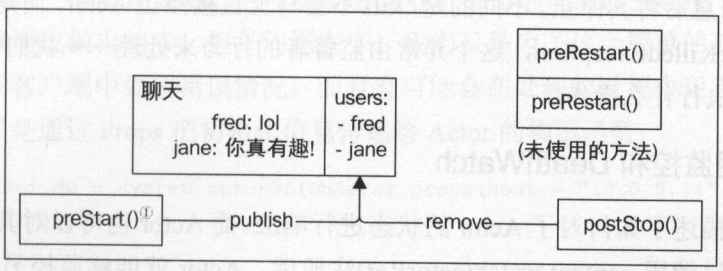


图 4-4

① 译者注：原书 postStart()有误，应改为 preStart()。

重启和停止时的消息处理

我们需要理解发生异常时的消息处理方式，并且根据应用程序采取合适的操作。这一点十分重要。我们可以定义监督策略，在抛出异常前重新尝试发送失败的消息，重试次数没有限制。也可以设置时间限制，比如最多重试 10 次或 1 分钟，只要达到其中任一限制就停止重试：

```
new OneForOneStrategy(2,
  Duration.create("1 minute"), PartialFunction)
OneForOneStrategy(
  maxNrOfRetries = 2) {
    case _: IOException => Restart
  }
)
```

一旦某个消息抛出了异常，该消息就会被丢弃，不会被重新发送。监督者会执行监督策略。在继续执行（resume）或是重启（restart）的情况下，就会处理下一条消息了。

如果有一个工作队列的话，我们可能会想要重试若干次，然后彻底返回失败。

终止或 kill 一个 Actor

有多种不同的方法可以用来停止一个 Actor。下面任一方法都可以停止 Actor：

- 调用 ActorSystem.stop(actorRef)；
- 调用 ActorContext.stop(actorRef)；
- 给 Actor 发送一条 PoisonPill 消息，会在 Actor 完成消息处理后将其停止；
- 给 Actor 发送一条 kill 消息，会导致 Actor 抛出 ActorKilledException 异常[①]。

调用 context.stop 或 system.stop 会导致 Actor 立即停止。发送 PoisonPill 消息则会在 Actor 处理完消息后将其停止。不同的是，kill 不会马上直接停止 Actor，而是会导致 Actor 抛出一个 ActorKilledException，这个异常由监督者的行为来处理——我们可以决定接收到这个异常时执行什么操作。

生命周期监控和 DeathWatch

监督机制描述了如何对子 Actor 的状态进行响应。而 Actor 也可以对其他任何 Actor 进行监督。通过调用 context.watch(actorRef)注册后，Actor 就能够监控另一个 Actor 的终止，而调用 context.unwatch(actorRef)就可以取消监控注册。如果被监控的 Actor 停止了，负责监控的 Actor 就会收到一条 Terminated(ActorRef)消息。

① 译者注：此处原书重复了第三条，有误。

安全重启

构建响应式系统时，要把失败处理列入设计考虑要素之中。我们要确保系统能够对失败情况做出正确的响应。笔者在实际项目中见到过的一个错误就是，启动 Actor 的时候没有处理失败情况，启动之后就对 Actor 的状态进行初始化。如果向 Actor 发送一些初始化消息，而消息中的信息对于 Actor 的状态至关重要，那么这些信息有可能会在重启的过程中丢失，一定要格外小心。

我们来看一个例子：假设表示数据库客户端的 Actor 没有构造参数，我们调用 tell，传入数据库的主机和端口来连接数据库。这看上去没有任何不合理之处——在 Java 中，我们经常会先创建对象，然后设置成员变量的值。我们也知道 Actor 能够安全地保存状态，所以这种用法似乎并没有什么问题：

```
ActorRef db = system.actorOf(DbActor.props());
db ! Connect(host = "10.0.8.14", port = "9001");

val actor = createActor(DbActor.props);
actor.tell(new Connect("10.0.8.14", "9001");
```

现在我们就有了一个 Actor，一旦这个 Actor 处理了第一条消息，就完成初始化，连接到了远程数据库。

假设我们先向这个 Actor 发送消息，然后由于网络可靠性的问题，Actor 发生了一个异常。此时 Actor 会重新启动。当 Actor 重启的时候，会调用 preRestart()，然后原来的 Actor 就会停止。这个时候原来 Actor 内的所有状态就消失了。接着会新建一个 Actor 实例，运行新建的 Actor 的构造函数，然后调用 postRestart()，Actor 就启动完成并运行了。

现在，用于连接数据库的消息就丢失了。我们可以编写一些额外的代码来处理这种情况。可以编写一个监督策略，也可以使用 Akka 生命周期的监控功能（DeathWatch）来对这些失败情况做出响应。但是问题在于，我们只是为了这个简单的初始化消息，就必须要立刻在客户端中处理错误情况，而且有可能会在处理的过程中犯下一些错误。更好的实现方式是通过 Props 把初始化信息传递给 Actor 的构造函数：

```
ActorRef db = system.actorOf(DbActor.props(host = "10.0.8.14", port =
"9001"));

val actor = createActor(DbActor.props("10.0.8.14", "9001"));
```

这样看上去更好：Actor 现在可以从构造函数中获取所有信息，并且自动连接。如果我们开始向 Actor 发送消息，然后 Actor 重启的话，由于构造函数的参数中包含了连接信息，因此 Actor 能够正确地从错误中恢复。不过如果 Actor 在初始化的过程中发生了异常

（比如说在尝试连接远程数据库的过程中），那么默认的监督机制会停止 Actor。所以，有一种情况是：如果 Actor 已经处在运行中，那么就能够安全地重新启动；但是如果无法完成初始化，我们就必须编写额外的代码实现监督策略，来处理 Actor 无法在启动时连接到数据库的情况。

很多时候，我们不希望 Actor 在初始化的过程中发生错误，而是会给 Actor 发送一条初始化消息，然后在 Actor 运行的过程中处理状态的变化。可以在 preStart()[①]中向 Actor 自己发送一条 Connect 消息来达到这一效果：[②]

```
public void preStart() {
    self().tell(new Connect(), null);
}

override def preStart = self ! Connect
```

这样一来，Actor 只在运行中才可能会出现错误，而且能够成功重启。所以就可以不断尝试连接数据库直到连上为止了。

仔细想一想，读者可能会觉得应该在 preRestart 方法中给自己发送 Connect (host, port) 消息。如果 Actor 出于任意原因丢弃了这条消息（可能是因为邮箱已满，又或者是由于重启导致了消息丢失），那么这条 Connect 消息还是可能会丢失。对于重要的 Actor 功能，使用构造函数来初始化更简单，也更安全。

上面只是快速而粗略地介绍了如何构造可能不断重新请求连接的 Actor。

有一个很重要的地方需要注意：上面的介绍中还没有涉及到一些细节和问题。首先，我们没有处理 Actor 在连接成功之前接收到的消息；其次，如果 Actor 长时间无法连接的话，邮箱可能会被填满。

我们将在本章中介绍状态处理。邮箱的问题将在“第 7 章：处理邮箱问题”中介绍。

4.3　状态

我们已经介绍过，Actor 能够安全地存储状态，它允许我们使用无锁的方式并发处理状态。现在我们就来介绍 Actor 如何通过不同的状态来改变它的行为。

我们可以使用几种不同的机制来改变 Actor 的行为：

- 基于 Actor 状态的条件语句；
- 热交换（Hotswap）：become()和 unbecome()；

① 译者注：原书为 postStart()，有误。
② 译者注：原书的代码有误，已更正。

- 有限自动机。

首先介绍一个基于键值数据库示例的例子，然后依次介绍上面的三种机制。

4.3.1 在线/离线状态

我们来回顾一下第一个分布式系统的误区。网络是不稳定的，而且试图通过网络互相通信的组件本身也有可能无法正常运行。我们希望应用程序能够从容地处理网络问题，而不会丢失太多的消息。

我们将继续开发数据库示例，改进远程客户端，尽可能从容地处理客户端/服务器模型下的网络分区问题，比如暂时性的网络组件故障或是数据库服务的重启。

我们在“第 2 章：Actor 与并发”中介绍了一个例子，在例子中有一个远程键值数据库，还有一个客户端可以连接这个数据库。我们使用 actorSelection 得到指向远程 Actor 的引用，然后开始向远程 Actor 发送消息。如果当我们启动客户端的时候，数据库还没有准备就绪，会发生什么呢？又或者如果数据库突然离线了，又会怎样呢？

转移状态

我见过最常用的使用状态的一个例子就是判断某个服务是否离线。我们在介绍失败处理的小节中探究了这个例子，了解了 Actor 重启时具体发生了什么。这里我们将继续使用通过数据库客户端连接远程数据库的例子。

在状态之间暂存消息（stash）

假设寿司餐厅的调酒师接到了一个啤酒订单，但是餐厅正在更换新的啤酒桶，那他就无法完成这个订单了。但是过一段时间，等新的啤酒桶到位了以后，就可以制作这杯啤酒了。所以调酒师不会因为啤酒桶没有准备好就直接拒绝这个订单，他会暂时把这个订单放一放，先去处理其他能够处理的订单（比如给厨师倒点儿米酒）。一旦啤酒桶准备就绪，他就可以重新开始处理啤酒的订单了。

和这个例子类似的是，Actor 也经常会处在一个无法处理某些消息的状态。如果数据库客户端离线了，那么在重新上线之前，它都无法处理任何消息。我们可以选择不断重新建立客户端的连接，直到连接成功为止。在这种情况下，在成功连接之前，客户端会丢弃收到的所有消息。另一种做法是把客户端无法处理的消息先放在一旁，等到客户端恢复连接状态之后再做处理。

Akka 提供了一种叫做 stash 的机制来支持这一功能。stash 消息会把消息暂存到一个独立的队列中，该队列中存储目前无法处理的消息：

```
if(cantHandleMessage) {
```

```
    stash();
} else {
    handleMessage(message);
}
```

unstash 则把消息从暂存队列中取出，放回邮箱队列中，Actor 就能继续处理这些消息了。

```
changeStateToOnline();
unstash();

To use stash in Java, your Actor will extend AbstractActorWithStash:
class RemoteActorProxy extends AbstractActorWithStash {
  [...]
}
To use stash in Scala, you mix in the stash trait:
class RemoteActorProxy extends Actor with Stash {
  [...]
}
```

现在我们就将使用 stash 机制来展示如何使客户端在在线和离线状态之间进行切换。

注意

要注意的是，虽然 stash()和 unstash()在希望快速改变状态的时候使用起来非常方便，但是 stash 消息的状态一定要和某个时间限制进行绑定，否则就有可能填满邮箱。

4.3.2　条件语句

最直观的方法就是将状态存储在 Actor 中，然后使用一个条件语句来决定 Actor 应该执行的操作。

我们可以存储一个布尔值，表示是否连接到数据库。如果已经连接，那么就试图处理消息；如果没有连接，就返回失败。

下面是使用 Java 存储状态并通过条件语句来处理消息的一个例子：

```
private Boolean online = false;
public PartialFunction receive() {
    return RecieveBuilder
        .match(GetRequest.class, x -> {
            if(online) {
                processMessage(x);
            } else {
```

```
            stash();
        }
    })
    .match(Connected.class, x -> {
        online = true;
        unstash();
    })
    .match(Disconnected.class, x -> online = false)
    .build();
```

下面是使用 Scala 来存储状态并通过条件语句来决定操作的代码：

```
var online = false
def receive = {
  case x: GetRequest =>
    if(online)
      processMessage(x)
    else
      stash()
  case _: Connected =>
    online = true
    unstash()
  case _: Disconnected =>
    online = false
}
```

这个条件语句是基于存储在 Actor 中的状态描述不同行为的最基本的方法。我们存储一个叫做 online 的布尔变量，表示 Actor 是否已经连接。如果已经连接，就处理消息。否则就像前面介绍过的那样 stash 消息。如果接收到一条 Connected 消息，就将状态修改为在线，并 unstash 所有消息。由于使用了 stash/unstash，所以一旦 Actor 在线，就会处理 stash 的所有消息。

很多时候，Actor 会存储状态，然后根据这个状态值的不同会有不同的行为。使用条件语句是一种非常过程化的用于处理行为和状态的方法。幸运的是，Akka 还提供了一些更好的选择。

4.3.3 热交换（Hotswap）：Become/Unbecome

使用条件语句的代码并不是很优雅，这种写法显然不是声明式的。Akka 提供了 become()和 unbecome()，用于管理不同的行为，这一用法可以大大改善代码的可读性。在 Actor 的 context()中，有两个方法：

- become(PartialFunction behavior)：这个方法将 receive 块中定义的行为修改为一个新的 PartialFunction。
- unbecome()：这个方法将 Actor 的行为修改回默认行为。

我们来看一下如何使用这个机制来优化前面的例子：

```
public PartialFunction receive() {
    return RecieveBuilder
        .match(GetRequest.class, x -> stash())
        .match(Connected.class, x -> {
            context().become(online);
            unstash();
        })
        .build();
}
final private PartialFunction<Object, BoxedUnit> online(
    final ActorRef another) {
        return RecieveBuilder
            .match(GetRequest.class, x -> processMessage(x))
            .build();
    }
```

下面是完成相同功能的 Scala 代码：

```
  def receive = {
    case x: GetRequest =>
      stash()
    case _: Connected =>
      become(online)
      unstash()
  }
  def online: Receive = {
    case x: GetRequest =>
      processMessage(x)
    case _: Disconnected =>
      unbecome()
  }
```

相较于条件语句，这种写法可读性更高。每个状态的行为都定义在自己独立的 PartialFunction 中。在 PartialFunction 中，使用模式匹配来定义不同的行为。这样我们就能够互不影响地阅读 Actor 中不同状态的行为。

Actor 一开始处于离线状态，这时候它无法对 GetRequest 做出响应，所以把消息 stash 起来。在接收到 Connected 消息之前，这些消息都会被暂存起来放在一边。一旦接收到 Connected 消息，Actor 就调用 become，将状态修改为在线（定义在 online 方法中）。此时，Actor 还会调用 unstash 将所有暂存的消息取回到工作队列中。这样就可以使用 online

方法中定义的行为来处理所有的消息了。

如果 Actor 接收到了一个 Disconnected 消息，就调用 unbecome，把 Actor 行为恢复为默认设置。要注意的是，我们可以定义任意数量的 receive 块，并相互切换。热交换基本上可以处理 Actor 的任何行为修改。这个抽象概念既优雅，又简单，可以写出可读性很高的 Actor。

stash 泄露

上面使用 stash 的例子有一个问题：如果要花很长时间才能接收到 Connected 消息，或者压根就收不到 Connected 消息，那么 Actor 就会不断地 stash 消息，最终导致应用程序内存耗尽，或者导致邮箱开始丢弃消息（后面的章节中会介绍邮箱）。无论何时，只要使用了 stash()，都最好设置一个限制，指定最长多少时间或者最多接收到多少消息后就不能再 stash 了。

最基本的方法就是调度执行一个超时消息，在一段特定的时间之后将这个消息发送给 Actor。

我们可以在 Actor 的构造函数或 preStart[①]方法中调度执行这个消息。接收到这个消息之后，就检查 Actor 是否已经连接。如果还没有连接，那么 Actor 就可以向上反映问题，由监督者来采取措施：

```
system.scheduler().scheduleOnce(
    Duration.create(1000, TimeUnit.MILLISECONDS),
    self(),
    CheckConnected,
    system.dispatcher(),
    null
);
```

如果 Actor 接收到消息时已经在线，那么就忽略该消息。如果 Actor 接收到消息处在离线状态，就会抛出一个异常：

```
.match(CheckConnected.class, msg ->
  throw new ConnectTimeoutException())
case _: CheckConnected => throw new ConnectTimeoutException
```

有限自动机（Finite State Machine，FSM）

还有另一个工具可以用来处理 Actor 中的状态：有限自动机（Finite State Machine，FSM）。和热交换很相似的是，FSM 中也有状态以及基于状态的行为变化。跟热交换比起来，FSM 是一个更重量级的抽象概念，需要更多的代码和类型才能够实现并运行。所以通常来说，热交换是一个更简单、可读性更高的选择。

① 译者注：原书为 postStart，有误。

我们已经有一个很好的在客户端中处理连接/断开连接的例子，我们将保留这个例子作为基本的客户端，然后寻找其他方法来优化客户端和服务器之间的通信。

本章一开始介绍了分布式计算的误区。其中有一点就是网络上的每个请求和响应都有延迟（误区：没有延迟）。因为每个请求和响应都有开销，所以可以通过减少发送消息的数量来降低总等待时间。为了优化客户端和服务器之间的通信，我们可以将多个消息组合到同一个请求中，减少网络上发送的消息数量。

使用传统的请求/响应协议时，应用程序首先会在网络上发起一个请求，然后等待响应。接收到响应后继续处理消息。在一个需要从数据库中读取多条记录的应用程序中，一个典型的数据流看上去可能会如下所示。

1. 客户端：通过网络发送 GetRequest(user)。
2. 服务器：向客户端返回 user 数据。
3. 客户端：等待 Success 响应。
4. 客户端：向服务器发送 GetRequest(article)。
5. 服务器：向客户端返回 article 数据。
6. 客户端：等待 Success 响应。
7. 客户端：返回 Success 响应，根据请求创建用户账户。

发送一个信号需要的时间加上接收响应的时间叫做往返延时（Round-trip Delay Time，RTD 或者 RTT）。有一个响应式准则就是尽快对用户做出响应，所以我们希望能够减少等待消息在网络上传输的时间。

一种优化我们的数据库 API 的方法就是允许数据库接收一个查询列表。这样我们就可以在同一个请求中包含多个操作：

```
ask(remoteActorSelection,
     Arrays.asList(
         new SetRequest(id, user, sender),
         new SetRequest("profile-" + id, profile, sender)),
     timeout);

 remoteActorSelection ? List(
   SetRequest(id, user, sender),
   SetRequest("profile-"+id, profile, sender)
)
```

注意到在上面使用列表的代码中，我们在消息中包含了一个 sender，这样就允许我们能完成 Future 或对 Actor 做出响应。

不过实践证明，在应用请求中创建一个单独的列表对于 API 的用户来说并不是很方便。我们可以改进客户端的 API 来解决这个问题。

我们可以在数据库客户端中创建一个 Actor，一旦累积接收到了多个消息，或是接收到了一个 Flush 消息，这个 Actor 就会把累积接收到的消息发送给远程数据库 Actor。这么做可以把收集和发送 Actor 的代码放到同一个 Actor 中，遵循了单一职责原则（Single Responsibility Principle）。（注意到这个例子和 Akka 文档中用于 FSM 的例子类似。如果需要一个可以快速查阅的手册或是对不同的实现有兴趣的话，读者可以查阅文档。）

FSM 中的类型有两个参数：状态和容器。我们将定义这两个参数，然后介绍如何构造 FSM。

定义状态

FSM 描述其状态的方式和我们已经介绍过的 Actor 有所不同。

在 FSM 的例子中，我们将对热交换的例子进行改进，不再 stash 消息，而是将消息存储在 Actor 中。

- **Disconnected**：离线，队列中没有任何消息；
- **Disconnected and Pending**：离线，队列中包含消息；
- **Connected**：在线，队列中没有消息；
- **Connected and Pending**：在线，队列包含消息。

在 Java 中，我们使用 enum 来定义状态：

```
enum State{
    DISCONNECTED,
    CONNECTED,
    CONNECTED_AND_PENDING,
}
```

在 Scala 中使用 case object 来定义状态：

```
sealed trait State
case object Disconnected extends State
case object Connected extends State
case object ConnectedAndPending extends State
```

定义状态容器

我们已经定义了状态，现在需要为 Actor 定义状态容器。状态容器就是存储消息的地方。FSM 允许我们定义状态容器，并且在切换状态时修改状态容器。所以由于我们要在改变状态的时候改变状态容器，因此非常适合在我们的例子中使用 FSM。

我们将保存一个请求列表作为状态容器。在接收到 Flush 消息的时候会处理这个列表中的请求：

```
public class EventQueue extends LinkedList<Request> {}
```

在 Scala 中，我们使用 type 定义来达到相同的目的：

```
object StateContainerTypes {
  type RequestQueue = List[Request]
}
public class Flush {}
case object Flush
```

现在就可以来定义 Actor 的行为了。

在 FSM 中定义行为

首先，FSM 必需要混入一个 Trait。

在 Java 8 中，我们继承 akka.actor.AbstractFSM<S, D>。

```
public class BunchingAkkademyClient extends AbstractFSM<State,
RequestQueue>{
    {//init block
    }
}
```

注意到我们创建了一个用于初始化的代码块，下文将在这个初始化块中定义行为。

在 Scala 中，我们实现 akka.actor.FSM[S, D]：

```
class BunchingAkkademyClient extends FSM[State, RequestQueue] {}
```

现在就可以在 Actor 中使用 FSM API 来定义不同状态的行为了。首先，调用 startWith 方法来定义 Actor 如何启动：

```
{
    startWith(DISCONNECTED, null);
}
startWith(Disconnected, null) //scala needs no init block!
```

接着，定义不同状态对不同消息的响应方法以及如何根据接收到的消息切换状态。有好几种方法来定义这些行为。用起来最方便的一种就是调用 when(S state, PartialFunction pf)。我们可以通过多次调用 when 语句来为每种情况都定义相应的行为。

在 Java 中，还是可以使用 Akka 提供的类似于 ReceiveBuilder 的工具来构造一个 PartialFunction。

我们将定义各个状态及其接收到 Flush、Connected 和 Request 消息时的响应：

```
when(DISCONNECTED,
    matchEvent(FlushMsg.class, (msg, container) -> stay())
        .event(GetRequest.class, (msg, container) -> {
            container.add(msg);
            return stay();
        })
        .event(Tcp.Connected.class, (msg, container) -> {
            if(container.getFirst() == null) {
                return goTo(CONNECTED);
            } else {
                return goTo(CONNECTED_AND_PENDING);
            }
        }));

when(CONNECTED,
    matchEvent(FlushMsg.class, (msg, container) -> stay()) {
        .event(GetRequest.class, (msg, container) -> {
            container.add(msg);
            return goTo(CONNECTED_AND_PENDING);
        }));

when(CONNECTED_AND_PENDING,
    matchEvent(FlushMsg.class, (msg, container) -> {
        container = new EventQueue();
        return stay();
    })
    .event(GetRequest.class, (msg, container) -> {
        container.add(msg);
        return goTo(CONNECTED_AND_PENDING);
    }));
scala.PartialFunction pf = ReceiveBuilder.match(String.class,
   x -> System.out.println(x)).build();
when(CONNECTED, pf);
```

在 Scala 中，我们使用 when(state) 在每个状态中通过模式匹配来定义 Partial Function：

```
when(Disconnected) {
  case (_: Connected, container: RequestQueue) =>
    if (container.headOption.isEmpty)
      goto(Connected)
```

```
      else
        goto(ConnectedAndPending)
    case (x: GetRequest, container: RequestQueue) =>
      stay using(container :+ x)
    }
  when (Connected) {
    case (x: GetRequest, container: RequestQueue) =>
      goto(ConnectedAndPending) using(container :+ x)
    }
  when (ConnectedAndPending) {
    case (Flush, container) =>
      remoteDb ! container;
      container = Nil
      goto(Connected)
    case (x: GetRequest, container: RequestQueue) =>
      stay using(container :+ x)
  }
```

我们在 Java 和 Scala 的例子中都定义了 3 个状态。Disconnected 状态会保存消息或是转移到 Connected 状态。它会忽略除了 Connected 和 GetRequest 以外的所有消息。

而在 Connected 状态中，我们只关心那些会将状态转移到 ConnectedAnd Pending 状态的消息。

最后，ConnectedAndPending 状态可以把容器中的所有请求都发送出去，也可以向容器中再添加一个请求。在 Scala 中，状态容器是一个不可变类型，所以在改变状态时其实传递的是一个新的容器。

注意到我们在某些状态中忽略了一些消息，而在另一些状态中则会处理这些消息。如果容器中没有请求消息的话，我们就会忽略 Flush 消息，不过如果容器中存在请求消息，我们就会处理接收到的 Flush 消息，处理并删除容器中的所有消息，然后转移回 Connected 状态。

Actor 必须要返回对状态的描述，要么是停留在 FSM 中的某个状态，要么是转移到了另一个状态。也就是说，和热交换比起来，FSM 要求 Actor 具有更强的表达能力。虽然使用 FSM 需要写一些模板代码，但是写出来的 Actor 代码在某些情况下也更清晰。如果要在 Actor 的不同状态中描述不同的行为，那么既可以使用热交换，也可以使用 FSM。读者可以根据自己的使用场景，衡量哪一种方案可以写出更可用的代码。很多情况下，热交换就已经足够了，也更简单，但是 FSM 是我们工具箱中的另一个工具，在某些情况下 FSM 写出的代码可维护性更高。

在这个代码块中要做的最后一件事就是调用 initialize()：

```
initialize();
```

4.3.4 通过重启转移状态

值得注意的是，Actor 没有定义任何 Disconnected 消息及其相关行为。如果在 Actor 的状态中遇到了异常，要切换回 Disconnected 状态，最简单的方法就是直接抛出一个异常，重启 Actor。很多时候，这就是用于处理异常情况最简单也是最可靠的方式。一旦 Actor 处于运行状态，就不要害怕重启 Actor！我们知道 Actor 系统中出问题的那部分会被直接重新创建。由于 Akka 抱着提高软件可靠性的态度，因此 Akka 团队的博客叫做“Let IT Crash”。如果抛出了异常，Akka 会负责处理，重新创建软件中发生错误的部分，无需开发人员担心。

要判断 Actor 是否连接，最好的方法就是每隔几秒钟向远程 Actor 发送一条消息。这个机制叫做“心跳”。我们可以通过心跳来判断 Actor 当前能否做出响应。如果无法响应的话，我们就可以重启 Actor，将异常信息输出到日志。在 FSM 的例子中，如果我们在几次心跳失败之后重启，那么 Actor 会丢弃所有尚未处理的消息。也正是因为这个原因，所以应用程序不会有太多的内存泄露。当远程数据库最终能够提供服务时，应用程序就会恢复正常的行为了。

4.4 课后作业

- 基于“第 2 章：Actor 与并发”中的 AkkademyDb，实现一个 Connected 消息：从客户端发送一个 Ping 消息，然后服务器返回一个 Connected 作为响应。
- 不使用 Ping 和 Connected 消息，改为在 preStart 方法中使用 Akka 的 Identify 消息来判断远程 Actor 是否能够提供服务。（参见文档。）通过这个方法而不是 ActorSelection 来得到远程 Actor 的 ActorRef。
- 仍然使用 Identity 和 ActorRef，在客户端中使用热交换来实现 Connected 状态。
- 每 2 秒钟发送一次 Identity 消息。如果连续两次没有收到响应，就重启 Actor。
- 如果不发送 Ping 消息并接收响应，有没有可能让客户端订阅每 2 秒钟发送一次的心跳消息呢？
- 更新客户端，使用 FSM 将多个消息打包到同一个请求中。你认为热交换和 FSM 哪个更适合这种情况？
- 实现监督策略，使 Actor 在每次重启时都将相关信息输出到日志（比如两次心跳失败导致的重启）。

4.5　小结

本章介绍了分布式系统的误区，帮助理解网络的一些特性及其对于应用程序可靠性和性能的影响。

由于 Actor 可能会遇到问题（无论是网络还是 Actor 内部状态都可能出问题），所以本章探究了 Actor 的生命周期。我们介绍了如何使用监督来对失败情况进行响应，以及如何自定义 Actor 的启动、停止和重启时候的行为。

我们还介绍了基于状态来改变 Actor 行为的不同方法。我们介绍了直接使用条件语句来实现基于状态的行为改变，然后学习了如何使用热交换和 FSM 来改进 Actor 的可读性。

现在，我们已经可以构建具备容错性的应用程序，能够对错误进行响应，并且明确地描述如何对不同的状态进行响应。

在下一章中，我们将介绍如何纵向扩展一个应用程序，更好地利用运行应用程序的硬件。

第 5 章
纵向扩展

纵向扩展指的是尽可能地利用单台机器上的硬件资源。使用 Akka，我们并不需要花太大的功夫就可以更简单地做纵向扩展，更好地利用硬件资源。事实上，要高效地利用硬件，一般来说不需要做太多的工作，但是必需要理解几项基本的技术。

在本章中，我们将介绍下面的话题：

- 多核计算的出现；
- 使用 Future 进行多核编程；
- 使用 Router 和 Actor 进行多核编程；
- 使用 Dispatcher 隔离性能风险。

要注意的是，本章虽然会讨论使用 Akka 来解决分布式问题，但是只会涉及到在单个节点上运行一个应用程序，不会涉及如何利用多台服务器来扩展应用程序。不过下一章会介绍横向扩展，讨论如何利用多台机器，以及如何使用 Akka 构建通过网络通信的应用程序集群。

5.1 摩尔定律

摩尔定律提到：每隔大约 18 个月，每平方英寸集成电路内的晶体管数量就会翻番。这和以前 CPU 的时钟速度每隔几年就会翻番有关。不过 CPU 时钟速度的增加的速度已经今非昔比了。

现如今，各种环境（比如服务器、计算机和手机）中的处理器都在以越来越多的 CPU 与核来吸引市场的眼球。例如，笔者的手机有 6 个核，工作用的工作站在 Xeon CPU 上有 12 个虚拟核，而 AWS 上的虚拟机最多有 40 个虚拟核。消费者硬件和服务器硬件的这些改变意味着我们在设计软件的时候需要考虑的因素也有了变化：如果应用程序只在单线程中运行，那么就算 PC 上有 8 个核，应用程序也只能高效地利用其中 1 个核。我们的应用程序需要能够利用这些资源——如果在单线程上运行应用程序，那么就只

能够利用 1 个核。现在，作为软件工程师，我们需要编写能够高效利用运行软件的硬件资源的并发软件。

“纵向扩展”这个术语的意思是：当我们给单个系统添加额外的资源（比如额外的 CPU 或内存）时，应用程序能够利用新添加的资源。如果有一个服务器，然后给应用程序提供了更多资源，那么应用程序在运行时能够利用新添加的资源。在云服务中，如果需要支持更高的负载，就可能会使用包含更多个核的更大的实例来代替原来的实例。如果应用程序能够利用新增的资源，那么就可以对其进行纵向扩展。（相反地，“横向扩展”指的是通过添加更多的机器或是 VM 来给系统增加资源，而不是通过使用更大的机器或 VM 来增加资源。）

由于现在大多数部署代码的环境都有多个可用的虚拟 CPU，因此我们需要改变编写代码的思路，尽量高效地利用硬件资源。作为一名现代开发者，要能够安全、高效地处理这些问题。因为我们现在可以使用更多的核，而不是单个计算速度更快的核，所以我们在设计应用程序的时候需要并行处理多个工作流，这样就能同时利用多个核了。

5.2　多核架构的分布式问题

在应用程序中利用多个核看上去似乎是个很困难的问题。使用传统的基于线程的抽象概念，很难正确地解决这个问题。如果多个线程都能够访问可变的共享状态，那么完成程序部署并大规模运行应用程序后，很容易就会发生资源竞争（race condition），无法得到正确的状态。Tim Sweeny 曾经说过，在并发的世界中，默认使用命令式编程是错误的。幸运的是，如今我们可以使用很多不同的抽象概念来利用所有可用的核。和使用同步与锁的线程抽象比起来，现在的这些抽象概念更容易用来编写正确的并发软件。

其实可以把如何进行多核架构看作是一个分布式问题：我们需要在另一个 CPU 或是另一台机器上完成一些任务。通过 Akka 来使用 Actor 时，纵向扩展和横向扩展之间的区别会开始显得不那么明显。我们可以忽略另一台机器和另一个核之间的区别，只把这个问题看作是向某个 Actor 发送一条消息。我们希望将一些任务发送到另一个地方来执行完成，然后在某一时刻接收对于发出请求的响应。可以将学习纵向扩展作为第一步，帮助我们理解如何最终进行横向扩展——如果可以在 8 个核上使用 Actor 完成工作，那么要在 8 台机器上使用 Actor 完成工作也八九不离十了。

本章的重点是介绍如何利用多个核而不是多台机器，所以会着重介绍这方面的细节。要利用多个核，最基本的机制就是并行：应用程序必须同时执行不同的操作。本质上来说，我们希望将工作分割成独立的子任务，然后使用不同的核同时运行这些子任务，这样就能利用所有可用的核了。

Akka 中提供了两种可以用来进行多核并行编程的抽象：Future 和 Actor。现在就来介绍如何使用这两种抽象来进行大规模并行计算。我们还会介绍如何使用这两种抽象来更高效地利用硬件资源。除此之外，还将介绍两者的适用场景。

5.3 选择 Future 或 Actor 进行并发编程

Actor 和 Future 这两个抽象都可以用于并发编程，那么究竟应该使用哪个呢？

我们从本书开始就已经开始使用 Akka 来创建可扩展的事件驱动系统，所以读者可能会想将 Actor 应用于所有的设计。也可能会觉得使用 Actor 是解决所有问题的正确方法。有句老话说得好：当你有了一把锤子以后，所有东西看起来都像是一个钉子。

实际上，要决定到底使用 Actor 还是 Future 其实并不简单。我曾经听别人说过一个通用准则："Future 用于并发，Actor 用于状态。"

换句话说，如果需要处理状态，那么可能马上就会想到用 Actor。如果不需要处理状态，只需要并发的话，那么可以试着用 Future。虽然这个准则还不错，但是它将问题过于简化了。在一些情况下，使用 Actor 会使得设计更易调试与维护。所以我们一定要具体问题具体分析，衡量两种方法的优劣，考虑设计是否简单。

5.4 并行编程

我们将从一个简单的例子开始展示如何使用 Actor 和 Future 来进行并行编程。我们需要多次执行一些非常耗时间的操作。本书在前面已经介绍了一个从网页解析出文章文本内容的例子。我们将再次使用这个例子，用 BoilerPipe 库来解析页面。解析文章的逻辑看上去如下所示：

```
public class ArticleParser {
    public static Try<String> apply(String html) {
        return Try.ofFailable(
            () -> de.l3s.boilerpipe.extractors
              .ArticleExtractor.INSTANCE.getText(html)
        );
    }
}

object ArticleParser {
  def apply(html: String) : String = de.l3s.boilerpipe.extractors
    .ArticleExtractor.INSTANCE.getText(html)
}
```

这就是需要并行执行的任务。我们将使用 Future 和 Actor 进行纵向扩展，更好地利用多核环境。

5.4.1　使用 Future 进行并行编程

Future 的可组合性很高，非常适合用来并行编程。我们在“第 2 章：Actor 与并发”中介绍过 Future 的函数组合技术，知道使用这种技术不需要花太大的功夫就可以安全地执行异步操作。我们可以使用 Future 来并行地处理集合中的元素，无需编写太多代码。

在下面的代码示例中，有一个叫做 articleList 的字符串列表，包含所有需要解析的文章所在的 HTML 页面。我们可以使用 Future 处理这个列表，达到高并发并高效利用系统资源。首先来看一下 Java 代码：

```
List<ComposableFuture<String>> futures = articleList
    .stream()
    .map(article -> CompletableFuture.supplyAsync(()
        -> ArticleParser.apply(article)))
    .collect(Collectors.toList());
Future<List<String>> articlesFuture = com.jasongoodwin.monads
    .Futures.sequence(futures).get();
```

注意到我们使用 better-java-monads 库来处理多个 Future，这个库中包含一个 sequence 方法，可以将一个 Future 列表转化成包含结果列表的单个 Future。要在项目中使用这个库，请将下面的依赖加入 build.sbt：

```
"com.jason-goodwin" % "better-monads" % "0.2.1"
```

在 Java 的例子中，我们将文章列表中的每篇文章 map 成一个 Future，这个 Future 的结果就是解析完成后的文章内容体。这样就可以并行地开始处理 articleList 中的所有文章。然后我们使用 sequence 将 Future 列表转换成单个 Future，结果是一个列表，其中包含所有文章内容体。使用 Scala 的例子也类似，不过更简洁。Scala 原生提供了 sequence 方法，用于对 Future 列表进行转换。

```
import scala.concurrent.ExecutionContext.Implicits.global
val futures = articleList.map(artlice => {
  Future(ArticleParser.apply(article))
})
val articlesFuture: Future[List[String]] =
  Future.sequence(futures)
```

我们对文章列表进行 map，然后通过 Future 来并行执行解析工作。接着使用 sequence 将 Future 列表转换成使用起来更方便的单个 Future，Future 中包含文章结果列表。通过上面的例子，我们可以看到，使用 Future 来进行并行编程是如此简单。

5.4.2 使用 Actor 进行并行编程

我们已经学习了如何使用 Future 来进行并行编程。现在就来看一下如何使用 Actor 来并行执行同样的任务。要注意的是，在我们的例子中，使用 Future 的代码非常简洁。而取决于具体情况的不同，单纯将 Actor 用于并发可能会需要编写更复杂的代码。下面就来学习如何使用 Actor 来完成相同的任务。

首先创建一个 Actor，负责调用 Future 例子中 ArticleParser 的静态 apply 方法。

下面是 Java 的 Actor 和消息：

```
public class ParseArticle {
    public final String htmlBody;
    public ParseArticle(String url) {
        this.htmlBody = url;
    }
}
public class ArticleParseActor extends AbstractActor {
    private ArticleParseActor() {
        receive(ReceiveBuilder
            .match(ParseArticle.class, x -> {
                sender().tell(ArticleParser.apply(x.htmlBody),
                    self());
            }
        ).build());
    }
}
```

下面是 Scala 的 Actor 和消息：

```
case class ParseArticle(htmlString: String)
class ArticleParseActor extends Actor {
  override def receive: Receive = {
    case ParseArticle(htmlString) =>
      val body: String = ArticleParser(htmlString)
      sender() ! body
  }
}
```

要并行完成任务，我们需要介绍“Router”的概念，用于将任务分发给不同的 Actor。接着我们就来介绍“Router”。

Router 介绍

在 Akka 中，Router 是一个用于负载均衡和路由的抽象。创建 Router 时，必须要传入一个 Actor Group，或者由 Router 来创建一个 Actor Pool。

注意 Group 和 Pool 这两个词的用法。为 Actor 创建 Router 时，一定要理解，有两种用来创建 Router 背后的 Actor 集合的机制。一种是由 Router 来创建这些 Actor（一个 Pool），另一种是把一个 Actor 列表（Group）传递给 Router。

在创建了 Router 之后，当 Router 接收到消息时，就会将消息传递给 Group/Pool 中的一个或多个 Actor。有多种策略可以用来决定 Router 选择下一个消息发送对象的顺序。

在我们的例子中，所有的 Actor 都运行在本地，我们需要一个包含多个 Actor 的 Router 来支持使用多个 CPU 核进行并行运算。如果 Actor 运行在远程机器上，也可以使用 Router 在服务器集群上分发工作，如图 5-1 所示。[①]

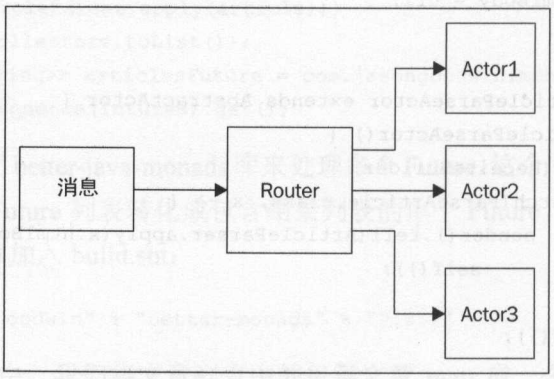


图 5-1

在我们使用本地 Actor 的例子中，可以选择 Actor Pool 的方式来创建 Router，由 Router 来创建我们所需的所有 Actor。在这种情况下使用 Router 非常简单：照常实例化一个 Actor，然后调用 withRouter，并传入一个路由策略，以及希望 Pool 中包含的 Actor 数量。Java 和 Scala 代码的逻辑相同：

```
ActorRef workerRouter =
    system.actorOf(
        Props.create(ArticleParseActor.class)
            .withRouter(new RoundRobinPool(8)));
```

① 译者注：原图有误，已更正。

```
val workerRouter: ActorRef =
  system.actorOf(
    Props.create(classOf[ArticleParseActor])
      .withRouter(new RoundRobinPool(8)))
```

也可以采用 Actor Group 的方式来创建 Router，传入一个包含 Actor 路径的列表：

```
ActorRef router = system.actorOf(new RoundRobinGroup(actors
    .map(actor -> actor.path()).props());
val router = system.actorOf(new RoundRobinGroup(actors
    .map(actor => actor.path).props())
```

现在，就有了将工作分发给不同 CPU 核所需的 Router 和 Actor。可以请求 Router 来处理列表中的每个消息，这样就可以并行执行操作了。我们将继续介绍 Router 的一些更高级的特性，深入理解 Router 的工作原理。

路由逻辑

注意到我们使用了 RoundRobinPool/RoundRobinGroup，用于指定 Router 将消息发送给各 Actor 的顺序。Akka 内置了一些路由策略，我们在这里将介绍一部分，如表 5-1 所示。对于一般情况来说，RoundRobin 和 Random 都是不错的选择。

表 5-1

路由策略	功能
Round Robin	依次向 Pool/Group 中的各个节点发送消息，循环往复。Random——随机向各个节点发送消息。
Smallest Mailbox	向当前包含消息数量最少的 Actor 发送消息。由于远程 Actor 的邮箱大小未知，因此假设它们的队列中已经有消息在排队。所以会优先将消息发送给空闲的本地 Actor。
Scatter Gather	向 Group/Pool 中的所有 Actor 都发送消息，使用接收到的第一个响应，丢弃之后收到的任何其他响应。如果需要确保能够尽快收到一个响应，那么可以使用 scatter/gather。
Tail Chopping	和 Scatter/Gather 类似，但是 Router 并不是一次性向 Group/Pool 中的所有 Actor 都发送一条消息，而是每向一个 Actor 发送消息后等待一小段时间。有着和 Scatter/Gather 类似的优点，但是相较而言有可能可以减少网络负载。
Consistent Hashing	给 Router 提供一个 key，Router 根据这个 key 生成哈希值。使用这个哈希值来决定给哪个节点发送数据。想要将特定的数据发送到特定的目标位置时，就可以使用哈希。在下章中，我们将讨论更多有关一致性哈希的问题。
BalancingPool	BalancingPool 这个路由策略有点特殊。只可以用于本地 Actor。多个 Actor 共享同一个邮箱，一有空闲就处理邮箱中的任务。这种策略可以确保所有 Actor 都处于繁忙状态。对于本地集群来说，经常会优先选择这个路由策略。

我们也可以实现自己的路由策略，不过大多数情况下并不需要这么做。

向同一个 Router Group/Pool 中的所有 Actor 发送消息

无论是使用 Group 还是 Pool 的形式来创建 Router，都可以通过广播，将一条消息发送给所有 Actor。例如，如果 Actor 都连接到一个远程数据库，运行中的系统由于发生了错误需要修改使用的数据库，那么就可以通过一条广播消息更新 Pool/Group 中的所有 Actor：

```
router.tell(new akka.routing.Broadcast(msg));
router !akka.routing.Broadcast(msg)
```

监督 Router Pool 中的路由对象

如果使用 Pool 的方式创建 Router，由 Router 负责创建 Actor，那么这些路由对象会成为 Router 的子节点。创建 Router 时，可以给 Router 提供一个自定义的监督策略。

创建 Router 的时候，可以调用 withSupervisorStrategy 方法指定 Router 对 Pool 中路由对象的监督策略。Scala 和 Java 的 API 是相同的。假设我们有一个叫做 strategy 的 SupervisorStrategy，那么可以使用下面的代码来创建 Router：

```
ActorRefworkerRouter = system.actorOf(
    Props.create(ArticleParseActor.class)
        .withRouter(new RoundRobinPool(8)
        .withSupervisorStrategy(strategy)));

val workerRouter: ActorRef =
  system.actorOf(
    Props.create(classOf[ArticleParseActor])
      .withRouter(new RoundRobinPool(8)
      .withSupervisorStrategy(strategy)))
```

由于使用 Group 方法创建 Router 的时候传入了事先已经存在的 Actor，所以没有办法用 Router 来监督 Group 中的 Actor。

监督 Pool 中的 Actor 是给 Router 指定监督策略的最常见的一种使用场景。除此之外，还有另一个场景会用到这种做法。如果有一个顶层的 Actor（使用 ActorSystem 的 actorOf 方法创建），那么这个 Actor 会将由守护 Actor 来监督。如果需要为这个 Actor 指定一个自定义的监督策略，那么一种方法就是创建另一个 Actor 来负责监督。除此之外，我们可以直接创建一个 Router，然后传递一个自定义的 SupervisorStrategy，由 Router 负责监督 Actor。由于这种方法不需要定义任何 Actor 行为，所以是最简单的为顶层 Actor 提供自定义监督策略的方法。

5.5 使用 Dispatcher

既然我们已经开始尝试优化应用程序的吞吐量和响应时间，那么就需要理解系统的所有瓶颈所在，了解请求/响应循环中时间都花在了什么地方。一旦给应用程序发送了任务之后，可用的线程就会试图处理所有的请求：我们需要理解如何去使用这些资源，这可以帮助服务使用尽可能少的延迟来达到尽可能多的吞吐量。

5.5.1 Dispatcher 解析

Dispatcher 将如何执行任务与何时运行任务两者解耦。一般来说，Dispatcher 会包含一些线程，这些线程会负责调度并运行任务，比如处理 Actor 的消息以及线程中的 Future 事件。Dispatcher 是 Akka 能够支持响应式编程的关键，是负责完成任务的机制。

所有 Actor 或 Future 的工作都是由 Executor/Dispatcher 分配的资源来完成的，如图 5-2 所示。

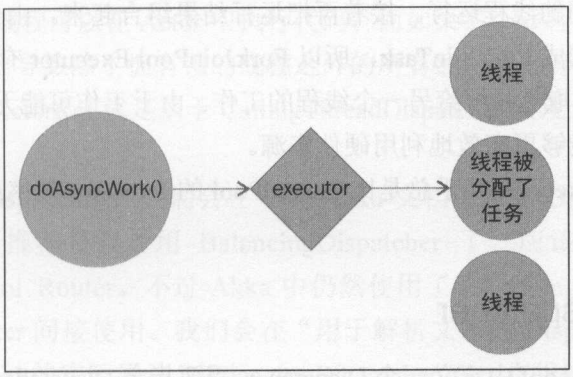


图 5-2

Dispatcher 负责将工作分配给 Actor。除此之外 Dispatcher 还可以分配资源用于处理 Future 的回调函数。我们会发现 Future API 接受 Executor/ExecutionContext 作为参数。由于 Akka 的 Dispatcher 扩展了这些 API，因此 Dispatcher 具备双重功能。

在 Akka 中，Dispatcher 实现了 scala.concurrent.ExecutionContextExecutor 接口，而这个接口又扩展了 java.util.concurrent.Executor 和 scala.concurrent.ExecutionContext。可以将 Executor 传递给 Java 的 Future，把 ExecutionContext 传递给 Scala 的 Future。

用于 Future 时，可以通过 ActorSystem 中的一个引用来获取 Dispatcher（ActorSystem.dispatcher）。可以在 ActorSystem 中通过 ID 查询得到配置文件中定义的某个 Dispatcher：

```
system.dispatcher //actor system's dispatcher
system.dispatchers.lookup("my-dispatcher"); //custom dispatcher
```

由于我们能够创建并获取这些基于 Executor 的 Dispatcher，因此可以使用它们来定义 ThreadPool/ForkJoinPool 来隔离运行任务的环境。稍后讨论什么时候这么做以及为什么要这么做。虽然要高效地使用 Executor，其实并不需要理解所有的细节，但是我们还是会先做一些介绍。

5.5.2　Executor

Dispatcher 基于 Executor，所以在具体介绍 Dispatcher 之前，我们将介绍两种主要的 Executor 类型：ForkJoinPool 和 ThreadPool。

ThreadPool Executor 有一个工作队列，队列中包含了要分配给各线程的工作。线程空闲时就会从队列中认领工作。由于线程资源的创建和销毁开销很大，而 ThreadPool 允许线程的重用，所以就可以减少创建和销毁线程的次数，提高效率。

ForkJoinPool Executor 使用一种分治算法，递归地将任务分割成更小的子任务，然后把子任务分配给不同的线程运行。接着再把运行结果组合起来。由于提交的任务不一定都能够被递归地分割成 ForkJoinTask，所以 ForkJoinPool Executor 有一个工作窃取算法，允许空闲的线程“窃取”分配给另一个线程的工作。由于工作可能无法平均分配并完成，所以工作窃取算法能够更高效地利用硬件资源。

ForkJoinPool Executor 几乎总是比 ThreadPool 的 Executor 效率更高，是我们的默认选择。

5.5.3　创建 Dispatcher

要在 application.conf 中定义一个 Dispatcher，需要指定 Dispatcher 的类型和 Executor。还可以指定 Executor 的具体配置细节，比如使用线程的数量，或是每个 Actor 一次性处理的消息数量。

```
my-dispatcher {
    type = Dispatcher
    executor = "fork-join-executor"

    fork-join-executor {
        parallelism-min = 2 #Minimum threads
        parallelism-factor = 2.0 #Maximum threads per core
        parallelism-max = 10 #Maximum total threads
    }
```

```
    throughput = 100 #Max messages to process in an actor before moving on.
}
```

有四种类型的 Dispatcher 可以用于描述如何在 Actor 之间共享线程：

- **Dispatcher**：默认的 Dispatcher 类型。将会使用定义的 Executor，在 Actor 中处理消息。在大多数情况下，这种类型能够提供最好的性能。
- **PinnedDispatcher**：给每个 Actor 都分配自己独有的线程。这种类型的 Dispatcher 为每个 Actor 都创建一个 ThreadPool Executor，每个 Executor 中都包含一个线程。如果希望确保每个 Actor 都能够立即响应，那么这似乎是个不错的方法。不过 PinnedDispatcher 比其他共享资源的方法效率更高的情况其实并不多。可以在单个 Actor 必须处理很多重要工作的时候试试这种类型的 Dispatcher，否则的话不推荐使用。
- **CallingThreadDispatcher**：这个 Dispatcher 比较特殊，它没有 Executor，而是在发起调用的线程上执行工作。这种 Dispatcher 主要用于测试，特别是调试。由于发起调用的线程负责完成工作，所以清楚地看到栈追踪信息，了解所执行方法的完整上下文。这对于理解异常是非常有用的。每个 Actor 会获取一个锁，所以每次只有一个线程可以在 Actor 中执行代码，而如果多个线程向一个 Actor 发送信息的话，就会导致除了拥有锁的线程之外的所有线程处于等待状态。本书前面介绍过的 TestActorRef 就是基于 CallingThreadDispatcher 实现支持在测试中同步执行工作的。
- **BalancingDispatcher**：我们会在一些 Akka 文档中看到 BalancingDispatcher。现在已经不推荐直接使用 BalancingDispatcher 了，应该使用前面介绍过的 BalancingPool Router。不过 Akka 中仍然使用了 BalancingDispatcher，但是只会通过 Router 间接使用。我们会在“用于解析文章的 Dispatcher”小节中介绍 BalancingPool 的实际用法。BalancingDispatcher 有一点很特殊：Pool 中的所有 Actor 都共享同一个邮箱，并且会为 Pool 中的每个 Actor 都创建一个线程。使用 BalancingDispatcher 的 Actor 从邮箱中拉取消息，所以只要有 Actor 处于空闲状态，就不会有任何 Actor 的工作队列中存在任务。这是工作窃取的一个变种，所有 Actor 都会从一个共享的邮箱中拉取任务。两者在性能上的优点也类似。

创建 Actor 的时候，可以给 Props 提供在 application.confg 中配置好的 Dispatcher 名称：

```
system.actorOf(Props[MyActor].withDispatcher("my-pinned-dispatcher"))
```

5.5.4　决定何时使用哪种 Dispatcher

本书现在已经介绍了如何创建 Dispatcher 和 Executor，但其实还不是很清楚要用它们来做什么。本章的目的就是介绍如何最大化地利用硬件资源，所以现在我们就将开始学习如何使用 Dispatcher 来编写对用户响应更快、对可能发生的性能问题更游刃有余的应用程序。

回到从网页中抽取、缓存并向用户返回文章内容体的例子。假设在这个例子中，需要从 RDBMS 获取用户的个人信息，而 JDBC 调用会阻塞线程。这个例子中线程的数量有限，而且会被阻塞，有助于强化解释本小节的内容。

进行纵向扩展的第一步是理解哪些情况的响应即时性最重要，以及对这些重要的请求做出响应时可能会发生资源竞争的地方。如果只使用默认的 Dispatcher，那么假如有 1000 个请求，其中 500 个会阻塞线程（如 JDBC），另外 500 个是非阻塞操作，那么我们不希望所有用于响应重要请求的线程都被阻塞操作占据，如图 5-3 所示。

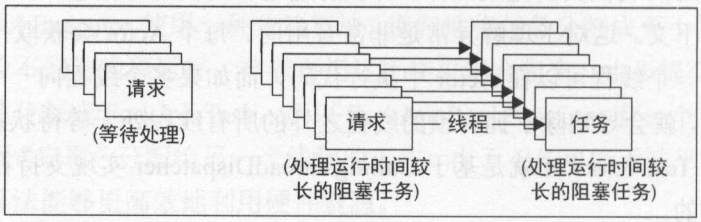


图 5-3

在图 5-3 中，我们简化了例子的规模，只展示了 8 个运行长任务（比如阻塞 IO 操作或是大量的 CPU 处理）的线程。从图 5-3 中可以看出，由于所有可用的线程都已经被占据，所以其他请求都必须要等待。直到有线程资源被释放，才可以开始处理其他请求。而处于等待中的请求可能只需要进行很快速的缓存读取，但是却必须要排队等待空闲资源。

这就说明了只使用一个资源池来随意分配工作可能会导致应用程序的某些很耗资源的操作占尽了所有资源，而最重要的基本使用场景却无法得到资源。

要改善这种情况，可以把用于运行高风险任务的资源和运行重要任务的资源隔离开来。如果我们新建一些 Dispatcher，把运行时间较长或是会阻塞线程的任务都分配给这些 Dispatcher，就可以确保应用程序的剩余部分仍然能够保持响应的即时性。我们希望能够把所有需要大量计算、运行时间较长的任务分离到单独的 Dispatcher 中，确保在糟糕的情况下仍然能够有资源去运行其他任务。

这样一来，我们就可以把应用程序执行长任务时的延时也隔离开来。如果 MySQL 出了问题，花了 30 秒钟才返回响应，那么至少应用程序的其他部分还是能够迅速地做出响应。

要采用这种方法，就必需首先分析应用程序的性能，理解应用程序在什么地方可能会阻塞线程，耗尽系统资源。我们需要对应用程序执行的任务进行分类。

在我们的例子中，可以对应用程序执行的任务分类如下：

- **解析文章**：运行时间较长的 CPU 密集型任务（10%的请求）
- **使用 JDBC 从 MySQL 读取用户信息**：运行时间较长、会阻塞线程的 IO 操作（10%的请求）
- **获取文章**：简单的非阻塞请求，从远程内存数据库中读取文章（80%的请求）

一旦我们知道了应用程序执行任务的类型，就可以开始理解这些任务是否可能引起性能问题——有没有可能会导致资源利用失控，影响应用程序的其他部分？分析阻塞线程和 CPU 密集型的任务，评估其是否可能导致应用程序其他重要部分无法获得资源：

- **解析文章**：如果有人提交了一些发布在网上的大型书籍，那么有可能所有线程都被占用，运行 CPU 密集型的长任务。这种情况风险中等，可以通过限制请求的大小来缓解。
- **使用 JDBC 读取用户信息**：如果数据库需要 30 秒才能进行响应，那么所有线程可能都会处于等待状态。这种情况风险更高。
- **获取文章**：读取文章的操作不会阻塞线程，不需要进行 CPU 密集型操作，风险较低。由于这种类型的操作占比最高，所以快速地对这类请求进行响应也很重要。

既然我们已经识别出了可能的高风险任务，那么当然希望使用单独的 Dispatcher 来负责这部分的工作，这样的话，发生问题的时候就不会对应用程序其他重要的部分产生负面影响。这是使用隔板法来隔离失败影响的另一个例子。不过在这里要提醒读者：对于变化可能对系统性能造成的影响，千万不要做任何假设，一定要实际测量。

我们已经事先对应用程序的工作和风险进行了分类，所以就可以把所有高风险任务分离出来，交给单独的 Dispatcher 来处理。使用的方法如图 5-4 所示。

在这个例子中，通过默认 Dispatcher 来处理使用 Akka 的非阻塞文章读取请求是没什么问题的。如果以后发现了其他风险，那么可以再进一步分割任务。不过刚开始使用默认 Dispatcher 就行了。

阻塞 IO（比如使用 JDBC 来读取用户信息）会有它自己的 Dispatcher，包含 50 或 100 个线程。会阻塞线程、等待 IO（使用 JDBC 驱动的数据库操作）的操作要和异步线程池分隔开，原因在于一旦所有线程都处于等待 IO 的状态，那么应用程序中的其他操作都无法继续执行。这可能是最重要的一点：千万不要把阻塞 IO 操作放在 Akka 的 Dispatcher 中。

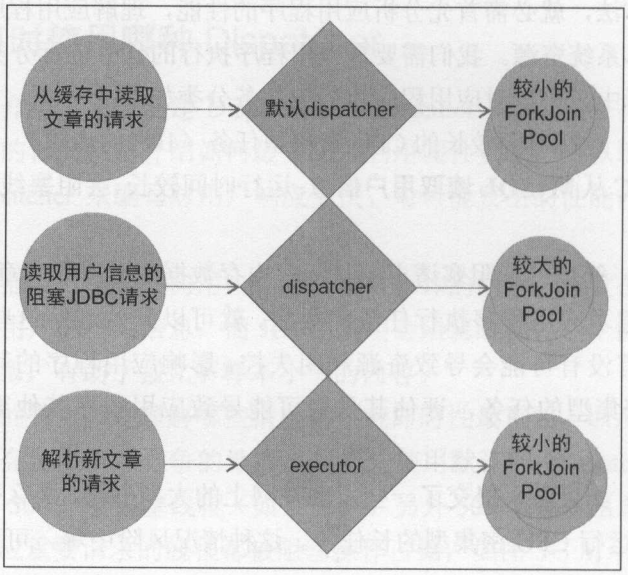


图 5-4

对于解析文章的请求，也由它自己的 Dispatcher 来处理，其中包含少量的线程。这里主要将 Dispatcher 用作隔离之用，以防用户提交的特别大的文章解析请求。如果提交的超大的请求，那么这个请求会导致队列中的请求都处于等待状态。所有为了处理这种极端情况，我们可以单独处理这类请求。在这种情况下，也可以使用 BalancingPool/BalancingDispatcher 将工作分配给 Pool 中所有用于解析文章的 Actor。除了能够提供隔离性以外，使用 BalancingPool 还有可能因为它原生的工作窃取特性改善资源利用效率。

5.5.5　默认 Dispatcher

有好多种使用默认 Dispatcher 的方法。既可以把所有工作都分离出去，只由 Akka 本身来使用默认 Dispatcher，也可以只在默认 Dispatcher 中执行异步操作，把高风险操作移到其他地方执行。无论怎么选择，都不能够阻塞运行默认 Dispatcher 的线程，而且要对于运行在默认 Dispatcher 中的任务多加小心，防止资源饥饿情况的发生。

要创建或使用默认 Dispatcher/ThreadPool 的话，其实不需要做什么。如果需要的话，只要在 classpath 内的 application.conf 文件中定义并配置默认 Dispatcher 即可。如下所示：

```
akka {
  actor {
```

```
    default-dispatcher {
      # Min number of threads to cap factor-based parallelism number to
      parallelism-min = 8
      # The parallelism factor is used to determine thread pool size using the
      # following formula: ceil(available processors * factor). Resulting size
      # is then bounded by the parallelism-min and parallelism-max values.
      parallelism-factor = 3.0
      # Max number of threads to cap factor-based parallelism number to
      parallelism-max = 64
      # Throughput for default Dispatcher, set to 1 for as fair as possible
      throughput = 10
    }
  }
}
```

我们可以在自己的 application.conf 文件中定义任意值，覆盖默认配置：

```
akka {
  actor {
    default-dispatcher {
      # Throughput for default Dispatcher, set to 1 for as fair as possible
      throughput = 1
    }
  }
}
```

默认情况下，Actor 完成的所有工作都会在这个 Dispatcher 中执行。如果需要回去 ExecutionContext 并在其中创建 Future，那么可以通过 ActorSystem 访问到默认的线程池，然后将其传递给 Future：

```
ActorSystem system = ActorSystem.create();
CompletableFuture.runAsync(() ->
    System.out.println("run in ec"), system.dispatcher());
val system = ActorSystem()
implicit val ec = system.dispatcher
val future = Future(() => println("run in ec"))
```

注意

对于在默认 Dispatcher 中的 Future 执行的操作要小心，这些操作会消耗 Actor 自身的时间。在下一小节中，我们将介绍如何解决这个问题。

在 Scala 中，扩展了 Actor 的类中已经包含了一个 implicit val 的 Dispatcher，所以在 Actor 中使用 Future 的时候就不需要再指定 Dispatcher 了。不过在 Actor 中使用 Future 的情况其实不是很多，要记住相对于 ask，应该优先使用 tell。所以如果发现有好多在 Actor 中使用 Future 的情况，那么可能需要衡量一下方法是否合理。

上面介绍了默认 Dispatcher 的用法及优化方法。现在我们就来看一下如何添加并使用其他 Executor。

5.5.6　使用 Future 的阻塞 IO Dispatcher

如果需要执行阻塞操作，那么不应该将这些操作放在主 Dispatcher 内执行，这样应用程序执行阻塞操作的时候仍然能够继续运行。

考虑下面这种用例：一个用户想要从某个 RDBMS 中获取用户信息，查看某篇文章的发布者。我们可能会有如下所示的服务接口：

```
import org.springframework.data.repository.CrudRepository;
import java.util.List;
public interface UserProfileRepository extends
    CrudRepository<UserProfile, Long> {
    List<UserProfile> findById(Long id);
}
```

在这个例子中，将使用 Spring Data，通过 JDBC 来访问数据库。Spring Data 只是随便选用的一个阻塞 API 的例子，可以用来以最少的代码说明阻塞 IO 的特点。但是也可以选择其他任何阻塞线程的方法。要查询一个使用 JPA 注解的 UserProfile，其实编写这个接口的代码就已经足够了。Spring 会通过分析方法名来提供具体实现。

在我们的例子中，有一个叫做 findById 的方法，接受一个 ID 作为参数，然后阻塞调用线程，等待从数据库返回的 IO 操作结果。

如果我们在一个 Actor 中调用这个方法，就会占据运行默认 Dispatcher 的线程，使得 Actor 无法再执行其他任何操作。

```
//Java
sender().tell(userProfileRepository.findById(id), self());
//Scala
sender() ! userProfileRepository.findById(id)
```

再提一下，如果接收到多个请求，这些请求都需要执行这段代码（可能由 Pool 中的 Actor 执行），那么所有线程都会处于等待状态，在有资源释放之前无法执行任何其他操作。

对于这种情况，最简单的解决方案就是使用另一个 Dispatcher，用另一些线程来执行阻塞操作。

首先，在 application.conf 文件中创建一个 Dispatcher，并且多给它分配一些资源：

```
blocking-io-dispatcher {
    type = Dispatcher
    executor = "fork-join-executor"
    fork-join-executor {
        parallelism-factor = 50.0
        parallelism-min = 10
        parallelism-max = 100
    }
}
```

这个 Dispatcher 最多可以在每个 CPU 核中创建 50 个线程，加起来一共最少 10 个线程，最多 100 个线程。对于一个配置合理、有索引的数据库来说，100 个线程的上限已经相当大了。

在对数据库执行阻塞 IO 操作时，如果发现某些查询运行时间很长，应该检查运行计划，优化数据库表的设计和查询语句，而不是增加线程。每个线程都有内存开销，所以不要随随便便地增加更多线程。只有已经对数据库表的查询、表以及分区进行了不断的评估、修改以及优化直至最优，才可以开始考虑修改线程池的大小。

既然我们已经配置好了一个 Dispatcher，现在就需要能够访问到这个 Dispatcher，然后在其中运行阻塞查询。可以在 ActorSystem 中查询，得到这个 Dispatcher 的引用。在 Actor 中，调用如下所示：[①]

```
//Scala
val ec: ExecutionContext = context.system.dispatchers.
lookup("blocking-io-dispatcher")
//Java
Executor ex = context().system().dispatchers().lookup("blocking-io-
dispatcher");
```

一旦得到了 Dispatcher 的引用，就可以使用该引用以及 Future API，在该 Dispatcher 中执行操作了。

```
//Java
CompletableFuture<UserProfile> future = CompletableFuture.supplyAsync(
    () -> userProfileRepository.findById(id), ex);
```

① 译者注：此处原书 Scala 和 Java 顺序发生错误，已更正。

```
//Scala
val future: Future[UserProfile] = Future {
  userProfileRepository.findById(id)
}(ec)
```

在 Scala 和 Java 的 Future API 中，我们只需要做一件事：把 Dispatcher 的引用作为 Future 的第二个参数传进去，Dispatcher 会负责剩余的所有工作。一旦有了结果，Future 就会完成。

有了 Future 引用之后，现在就可以照常执行正常的 Actor 操作：比如使用 patterns. Pipe 将结果异步地发送给另一个 Actor。

这种 Future 的用法是一种非常重要的技术：阻塞 IO 很快就会影响应用程序的性能。推荐读者试着直接使用非阻塞的数据库驱动，不过如果已经选择了需要使用阻塞驱动的技术，那么这是一个不错的方法。

和处理阻塞 IO 的方法类似，也可以使用 Future 来获取计算密集型任务的结果，将执行计算的过程移到另一个 Dispatcher 中，确保 Actor 能够继续执行。

5.5.7　用于解析文章的 Dispatcher

下面是最后一个例子，介绍如何将 Actor 分配给另一个 Dispatcher。这和 JDBC 的例子有所不同。在 JDBC 的例子中，只是将某个任务交给另一个 Dispatcher 来运行，而在这个例子中，我们实际上是把 Actor 完全交给另一个 Dispatcher，而不仅仅是 Actor 中某些任务。这种做法适用于任何任务负载较重的 Actor。

我们将介绍两种可用的方法：

- 定义一个 Dispatcher，用于 Actor Pool；
- 使用 BalancingPool Router（Router 内部使用了 BalancingDispatcher）。

在 Actor 中使用配置好的 Dispatcher

现在我们将介绍如何在 application.conf 中配置一个 Dispatcher，以及如何在创建 Actor 时将 Actor 分配给配置好的 Dispatcher。做法是相当直接的，和我们目前为止已经介绍过的许多其他操作差别不大。

首先，我们将在 application.conf 中创建另一个 Dispatcher，用于解析文章。这次分配的线程数量少一些。

```
article-parsing-dispatcher {
    # Dispatcher is the name of the event-based dispatcher
    type = Dispatcher
    # What kind of ExecutionService to use
```

```
    executor = "fork-join-executor"
    # Configuration for the fork join pool
    fork-join-executor {
        # Min number of threads to cap factor-based parallelism number to
        parallelism-min = 2
        # Parallelism (threads) ... ceil(available processors * factor)
        parallelism-factor = 2.0
        # Max number of threads to cap factor-based parallelism number to
        parallelism-max = 8
    }
    throughput = 50
}
```

现在，要创建 Actor 并将其分配给刚配置好的 Dispatcher，只要在创建 Props 的时候直接调用 withDispatcher 方法即可。我们将使用一个范围来创建一个 Actor 列表，然后将这些 Actor 分配给 Dispatcher。

```
//Java
List<ActorRef>routees = Arrays.asList(1,2,3,4,5,6,7,8)
    .stream().map(x ->
        system.actorOf(Props.create(ArticleParseActor.class)
            .withDispatcher("article-parsing-dispatcher"))
    ).collect(Collectors.toList());
//Scala
val actors: List[ActorRef] = (0 to 7).map(x => {
  system.actorOf(Props(classOf[ArticleParseActor])
  .withDispatcher("article-parsing-dispatcher"))
}).toList
```

现在就可以使用通过这种方法创建的 Actor 做任何想做的事了。例如，我们可以创建一个使用这些 Actor 的 Router，这样就能够轻松地使用这些 Actor 执行并行操作了。

```
//Java
Iterable<String> routeeAddresses = routees
    .stream()
    .map(x -> x.path().toStringWithoutAddress())
    .collect(Collectors.toList());
ActorRefworkerRouter = system.actorOf(new RoundRobinGroup(
    routeeAddresses).props());
//Scala
val workerRouter = system.actorOf(RoundRobinGroup(
  actors.map(x => x.path.toStringWithoutAddress).toList).props(),
    "workerRouter")
workRouter.tell(
  new ParseArticle(TestHelper.file), self())
```

这和我们之前看到过的在 Router 内创建 Actor 的语法略有不同。我们看到过使用 Pool 来创建 Router 的例子：由 Router 负责创建作为路由对象的 Actor。在这个例子中，我们将提前创建好的 Actor 作为一个使用某种负载均衡策略的 Group（RoundRobinGroup）传入。Router Group 接受一个地址列表作为参数，然后生成创建 Router Actor 所需的 Props。这和我们处理远程 Actor 的方法非常类似。

注意

如果也想把 Router 分配给某个 Dispatcher，那么 Group 也可以接受 Dispatcher 的名字作为参数。

使用 BalancingPool/BalancingDispatcher

由于 Actor 是本地的，所以我们有比前面使用的 RoundRobinGroup 更好的选择。对于本地 Actor 来说，可以使用 BalancingPool 来创建一个 Router，本章早些时候曾经简要地介绍过。使用 BalancingPool 时，Pool 中的所有 Actor 会共享同一个邮箱，然后通过高效的工作窃取机制将任务重新分配给任何空闲的 Actor。由于共享同一个邮箱，因此使用 BalancingPool 有助于确保在有工作的时候降低 Actor 的空闲率。由于 Router 并没有像 ForkJoinPool 那样重新分配工作，只是由空闲的 Actor 从共享邮箱中抽取下一条消息处理，所以从技术上来讲并不是工作窃取。最后的效果是一样的：不可能发生某个 Actor 的工作队列中有多条消息而另一个 Actor 处于空闲状态的情况。因为我们能够确保更多的 Actor 处于工作状态，所以这种做法通常都能获得比其他负载均衡策略更高的资源利用率。

BalancingPool 使用了一种特殊的 Dispatcher：BalancingDispatcher。在大多数情况下，我们希望 BalancingDispatcher 中的线程数量和使用的 Actor 数量相等。

首先，我们在 application.conf 中配置默认的 BalancingDispatcher 的 Executor，使用正正好好 8 个线程：

```
//Dispatcher for BalancingPool
pool-dispatcher {
   fork-join-executor { # force it to allocate exactly 8 threads
     parallelism-min = 8
     parallelism-max = 8
   }
}
```

然后，我们创建一个包含 8 个 Actor 的 Pool——Actor 的数量与 Dispatcher 中线程的

数量相同。

```
//Java
ActorRef workerRouter = system.actorOf(new BalancingPool(8)
    .props(Props.create(ArticleParseActor.class)),
        "balancing-pool-router");
//Scala
val workerRouter = system.actorOf(BalancingPool(8).props(
  Props(classOf[ArticleParseActor])), "balancing-pool-router")
```

在本地工作时，这是一种在所有 Actor 之间平衡负载的绝佳方法。

5.5.8 并行最优化

要确定硬件上的最优并行度，方法只有一种：测试。如果没有真正进行测试以及调整，那么关于时间花在什么地方以及改变如何影响系统的所有猜想基本上都是错的。

如果使用的线程数量远远多于处理器核的个数，那么也会降低性能。所以不要觉得线程越多越好，选择包含很多 Actor 的 Pool。Akka 需要在 Actor 之间来回切换，而 Executor 也需要在线程之间平衡负载。操作系统需要进行调度，将 CPU 时间分配给运行中的线程，并通过交换运行线程与等待线程的状态进行上下文切换。也正是由于这些开销，最优的并行处理能够达到的处理时间实际上可能比我们想象的要长。要知道某个改变带来的影响，唯一的办法就是测试！

5.6 课后作业

- 使用 Future 和 Actor 练习执行并行操作。
- 试着将任务分配给不同的 Dispatcher。
- 写下一些关于使用不同线程数量能达到的性能好坏的猜想。
- 在高负载之下测量系统的性能。对于同一个测试测量多次，取平均值。增加和减少线程以及 Actor 的数量对系统性能有怎样的影响呢？
- 猜想和测量结果有何出入？
- 基于实验观察结果，对于并行最优化给出一些通用的建议。

注意

使用的 CPU 核比较多的时候更容易进行总结。我们可能会发现测试结果出乎意料。

- 能否确定上下文切换的次数？这对性能有什么样的影响？在 Linux 系统上，我们

可以在/proc/[pid]/status 中找到信息。我们可能需要使用 ps 寻找正确的运行 Java 程序的 pid。

- 思考一下最近参与开发的一款应用程序——如何将工作分配给不同的 Dispatcher？
- 对于每个 Dispatcher，应该使用多大的线程池？

5.7　小结

在本章中，我们介绍了两个主要的概念：第一，纵向扩展本地应用程序，使之更好地利用多核硬件；第二，将工作分割，交由单独的 Dispatcher 来处理，使得应用程序的某些部分发生性能问题时，其余部分与之隔离，不受影响。

要进行纵向扩展，就需要并发执行操作。所以我们介绍了一些用于并行工作的技术。我们研究了如何使用 Future 和 Actor 来执行并行操作。接着，介绍了如何对一个应用程序进行分析，确定如何将工作分割，将不同类型的工作（比如计算密集型操作和阻塞 I/O 操作）交给不同的 Dispatcher 来处理。因为单独的 Dispatcher 资源耗尽时，不会影响其他 Dispatcher，所以我们可以确保应用程序的其余部分仍然能够及时做出响应。

在下一章中，我们将在本章所介绍的概念的基础上，学习如何开始将工作负载分配到多台机器上执行。

第 6 章
横向扩展——集群化

在上一章中，我们介绍了如何通过并行操作来提高单个物理主机上的硬件利用率。还介绍了如何在单台主机上通过多个特定的 Dispatcher 来隔离性能问题。本章将介绍单台物理主机达到资源上限时的情况以及如何使用多台机器来处理工作。这听起来复杂得多，但是 Akka 提供的工具使得我们可以快速上手。

在本章中，我们将讨论下面的话题：

- 分布式系统中使用的一些基础概念
- Akka Cluster 的介绍
- 使用 Akka Cluster 来构建分布式系统

6.1 Akka Cluster 介绍

本书已简要介绍过 Akka Remoting。理解 Remoting 对于学习 Akka 在网络上的通信方式很有帮助，不过在本章中，我们将使用 Akka Cluster 机制，而不是简单的点对点通信来提高部署系统的容错性和灵活性。本小节介绍一些集群的概念，以及 Akka Cluster 如何管理节点，支持应用程序按需横向扩展。

首先，我们来定义什么是集群。维基百科上的定义是："计算机集群由一系列间接或直接连接的计算机组成，这些计算机共同完成工作，从很多角度上来说，可以将这些计算机看作是单个系统。"具体到本章中对于集群的定义，我们可以认为一个集群就是一组机器（也可能是虚拟机），我们将这些机器称作节点（node）或是成员（member），同一个组中的所有节点遵循某个协议。

6.2 巨型单体应用 vs 微服务

作为一个开发者，从笔者开始写代码起，Martin Fowler 就一直是笔者心目中的英雄

之一。每当看到他分析某个问题时，也会跟着思考，记笔记。

如果我们在谈论微服务和分布式系统时，读者可能会想读一下他的网站 http://www.martinfowler.com/上的一些相关文章。

第一个有价值的观点是：在一个应用程序的生命周期中，当应用程序的复杂度还比较低的时候，和试图构建网络服务比起来，由一个小团队来构建单个大一点的应用程序（单体应用）可能效率更高。一开始可以先编写单个应用程序。随着应用程序变得越来越复杂，由于团队之间需要越来越多的协作来合并代码以及共同开发功能，生产效率会下降。这个时候，就可以开始看到使用小型网络服务给生产效率带来的好处。

第二个有价值的观点是：当我们需要将服务切分成许多更小的应用时，首先编写单体应用程序可以帮助我们理解应该将哪些部分分割出来。Fowler 认为如果没有真正开发系统一段时间的话，很有可能会选择把错误的服务实现成单独的应用。

一旦有了些在生产环境中运行系统的经验和数据，要把哪个部分分离出来才能够在新增功能时带来性能和团队开发速度的好处就会比较清楚了。开始部署微服务之后，我们就可以用一种非平衡的方式对负载更高的服务进行扩展，给应用程序中使用更多的部分分配更大的服务器集群。

幸运的是，使用 Akka 时，更多时候只需要通过配置选项而不是代码就可以做这些部署决定。而使用多个代码库和让一个大团队使用同一个代码库比起来，也要简单得多。使用多个代码库解决了布鲁克斯定律（Brooks Law，给一个已经延期的项目添加资源会使其延期的时间更长）提到的问题，原因在于它减少了通信渠道，把代码提交分隔到不同的代码库中（不会再有合并代码带来的噩梦）。

而当单体应用变得极度复杂，难以管理，除非是有很好的基于数据支持的原因（性能或团队效率相关），否则最终我们很可能会考虑选择构建一个分布式系统，而不采取单体应用的方式来部署。

无论是选择构建单体应用还是微服务，我们可能都想从一开始就把不同的功能模块放在不同的库中。即使还没有真正准备部署多个独立的应用程序，让不同的团队使用不同的代码库也是有益的。一旦想要分割应用程序，如果一开始就使用多个代码库，那么管理起共享代码来要简单得多。

6.3　集群的定义

集群就是一组可以互相通信的服务器。集群中的每台服务器成为一个节点或成员。集群可以动态修改大小，并且在发生错误情况时继续运行，把影响降到最低。所以集群需要具备两个功能：失败发现以及使得集群中的所有成员最终能够提供一致的视图。

6.3.1 失败检测

随着我们向集群中添加节点，节点很可能会发生错误，某些网络分区也可能暂时不可访问。因此集群是一个动态的实体，在服务器宕机或是不可用时会变小，而在添加服务器之后集群会变大（比如处理更高的负载）。集群中的节点通过互相发送消息来确定对方是否可用。节点基于能否获得其他节点的响应来确定其他节点是否可用。

如果集群中的每台服务器都需要和其他所有服务器进行通信，那么集群的性能不会随着节点的增加而线性提高。原因在于每增加一个节点，需要的通信开销都会指数增加。为了降低监控其他节点健康程度的复杂度，Akka 中的失败检测只会监控某个节点附近特定数目的节点。例如，在一个由 6 个节点环形排列组成的节点中，每个节点只会监控它后面的 2 个节点是否发生错误。在 Akka Cluster 中，每个节点监控的最大节点数默认是 5 个节点，如图 6-1 所示。

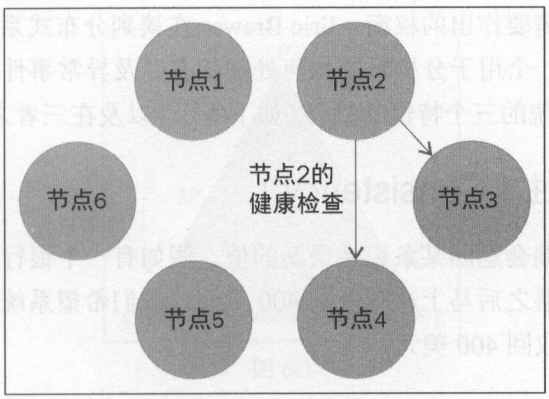


图 6-1

在 Akka 中，失败检测是通过在节点间发送心跳消息并接收响应来完成的。Akka 会根据心跳的历史记录以及当前的心跳信息计算出某个节点可用的可能性。Akka 会依照这些数据和配置的容错限制得到计算结果，然后将节点标记为可用或不可用。

6.3.2 通过 gossip 协议达到最终一致性

思考一下失败检测的处理，集群中的每个节点和它的邻居节点互相交换已知的状态。接着，这些邻居节点再将已知的状态传递给它们的“邻居”，以此往复，直到某个节点的已知状态传递给了集群中的所有成员。节点会仔细计算并得到集群中其他成员的状态。

如果某个节点被另一个节点访问，那么集群中的所有节点都会认为这个节点不可用。

这种最终把状态传递给整个集群的机制叫做 gossip 协议或是 epidemic 协议（因为信息在集群中的传播就像病毒一样！）。许多最终一致性的数据库（比如 Riak 和 Cassandra）的实现方式都非常类似。在这些数据库中，Amazon 发表的关于 Dynamo 的论文影响力非常大。

我们暂时不需要知道太多 Akka Cluster 的内部原理，只需要知道 Akka 会负责确定集群中是否有状态变化，并且负责将发生的任何变化传递给集群中的所有节点。

本章剩余的章节会构建自己的集群。在这个过程中，我们将介绍许多集群的工作原理细节。如果读者想要了解更多关于 Akka Cluster 工作原理的背景知识，在 Akka 的文档中有一个叫做 Cluster Specification 的文档，可以去查阅。

6.4　CAP 理论

我们将在本章中介绍一个无状态的工人的例子，还将介绍如何在集群中存储状态，理解在做设计选择时需要作出的权衡。Eric Brewer 在谈到分布式系统中的妥协时经常引用 CAP 理论，这是一个用于分析系统如何处理状态以及异常事件的非常有用的模型。CAP 理论是分布式系统的三个特性的缩写（如下所示）以及在三者之间进行选择的权衡。

6.4.1　C –一致性（Consistency）

一致性是指客户端会返回某条记录最新的值。假如有一个银行账号，如果我们在存入一张 400 美元的支票之后马上试图取回 400 美元，我们希望系统能够给出正确的账户余额，并且允许我们取回 400 美元。

6.4.2　A –可用性（Availability）

可用性是指一个没有发生失败的节点能够返回一个合理的响应。（例如，给出某个写操作是否成功的精确描述）。

6.4.3　P –分区容错性（Partition Tolerance）

分区容错性指的是如果某个节点由于暂时的网络错误而被从网络中移除，那么系统可以继续正常运行。如果数据被冗余备份到三个节点，那么如果其中一个节点暂时变得不可用，而另两个节点仍然能够正常运行[①]，那么就认为系统具备分区容错性。

① 译者注：此处原文表述有误，已更正。

6.4.4 CAP 理论中的妥协

CAP 理论通常都会表述如下：在一致性、可用性和分区容错性这三个特性中，一个分布式系统只能够选择满足其中两个。这种表示过于简单，可能会让人误解，如果阅读过一些文章的话，可以看到许多持不同观点的争论。

首先，假设我们希望系统具备分区容错性，只在可用性和一致性之间进行妥协。读者可能会问为什么。在接收到一个有超时限制的请求时，如果节点不可用，我们其实就需要在两种方案之间进行选择：要么返回错误（选择一致性），要么继续，即使服务器之间可能会不一致（选择可用性）。等待的时间过长会导致该请求被抛弃，所以时间是一个重要的因素，系统必须要在上面两者中做出决定，如图 6-2 所示。我们将介绍更多相关知识，不过 Eric Brewer 在发表了 CAP 理论的论文 12 年后，又写了一篇文章，对这一点做了深入讨论。参见 http://www.infoq.com/articles/cap-twelve-years-later- how-the-rules-have-changed。

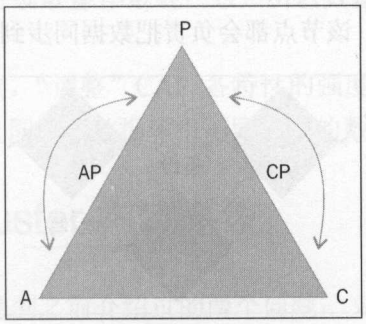


图 6-2

CP 系统——一致性优先

要实现一个一致性的分布式存储，有很多种不同的方法。一个最简单的强一致性数据库的例子也许有一个 master 节点和任意数量包含冗余备份的附属节点。数据永远会在 master 节点写入，而如果要确保读取最新的数据，那么页必须要从 master 节点读取。如果 master 节点发生错误，那么系统就不再可用，如图 6-3 所示。

通常来说，可以采取一些应对错误的机制，比如让某个附属节点成为新的 master 节点。因为 master 节点发生故障的时候我们无法从系统中读取数据，也无法写入数据，所以我们放弃了可用性，而是启动错误处理过程，选出新的 master 节点。一旦错误处理过程完成，系统的可用性就恢复了。Redis Sentinel 和具备冗余备份的 RDBMS 都是强一致性分布式系统的很好的例子。

图 6-3

如果需要原子读取/写入、事务等支持的话，那么可以选择具备一致性的系统。

AP 系统–可用性优先

选择支持可用性和分区容错性，并牺牲一致性的系统被称为具有“最终一致性”。在高可用的分布式数据库（如 Cassandra 和 Riak）中，这是一种非常常见的模型。

由于分析 AP 系统稍微复杂一些，因此我们将在本章中介绍更多 AP 系统的细节。不过我们可以先看一下 AP 系统的一种可能的实现例子。假设在 3 个节点上保存了数据的 3 份备份。当我们写入数据时，数据会被写入到其中 1 个节点，然后会被复制到其余 2 个节点。无论写入的是哪个节点，该节点都会负责把数据同步到其他节点，如图 6-4 所示。

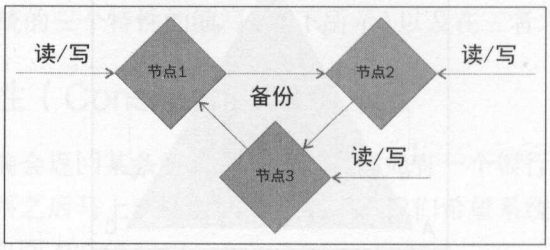


图 6-4

当我们从一个节点读取数据时，只需要访问一个节点就够了。所以客户端可以任意选择一个节点读取。在这个例子中，因为从某个节点读取数据时，数据可能并不是最新的（不具备一致性），所以我们的系统具备最终一致性。但是由于我们可以从任何一个节点读取数据，也可以向任何一个节点写入数据，所以系统具备分区容错性和高可用性。如果某个节点变得不可用，我们可以直接尝试使用另一个节点。

注意

要注意的是，这一点很难正确做到。要确定系统中事件的时间和顺序是非常有挑战性的。不同机器上的时间是无法完全同步的，所以需要采取其他方法来处理顺序问题（比如向量时钟）。

灵活的一致性程度

从实际应用的角度来说，在 3 个特性之间的权衡并不是非黑即白的，其实可以平缓地过渡。例如在一个最终一致性系统中，如果想查询某条记录，而该记录有 3 个冗余备份，那么我们可以选择从任意一个节点读取数据，这样一致性就比较弱。我们也可以选择其中任意两个节点返回数据，这样一致性就提升了。同样，我们可以向所有 3 个节点请求返回结果，这样就能提供最高的一致性。当我们从 3 个节点中得到返回数据时，有多种不同的机制可以用来对返回的记录排序，我们可以选择最新的数据。但是，这么做会牺牲分区容错性。如果我们需要所有 3 个备份都可用，那么就无法容忍其中任何一个消失。很多时候，只要求特定数量的节点或是大多数节点可用并且能返回一致的结果是在一致性和分区容错性中进行权衡的一个不错的方法。

同样，在一个 CP 系统中，我们可以允许从附属节点读取数据，牺牲一部分一致性来获取更高的可用性。如果保持仍然只能向 master 节点写入数据，那么我们还是有高一致性的写入操作，但是允许读取操作最终一致，所以数据库在读取操作上就变成 AP 系统了。

我们可以根据具体的用例，“调整”CAP 各特性的强度，使之最适合用例的需要。甚至可以对同一个应用程序、同一个数据库中不同类型的数据混合使用这些策略！

6.5 使用 Akka Cluster 构建系统

在本章中，我们将再次遇到之前介绍过的两个问题：文章解析和键值存储。不过我们将在本章中介绍如何使用 Akka Cluster 将这两个问题分布到多台服务器上处理。假设我们已经构建了一个应用程序，这个程序需要能够在多个节点上扩展，并且具备高可用性，我们将学习到如何通过添加更多的节点来扩展我们的系统。

读者可能无法一口气就编写出 Cassandra，不过我们将在本章中编写出一个能够横向扩展的分布式服务。也会顺便介绍在分布式解决方案中的一些常用技术。

6.5.1 创建集群

在本小节中，我们将介绍如何使用 Akka 的 Cluster 模块创建一个集群。本书前面简要介绍过 Remoting，现在我们就将更详细地学习集群的知识。Akka Cluster 基于 Remoting，但是更强大，用处也更大。如果使用 Remoting，那么我们需要在基础设施或者代码中自己考虑高可用性这样的问题。而 Akka Cluster 会负责解决很多这些问题，因此是我们构建分布式 Actor 系统的绝佳选择。

项目配置

首先，我们需要对 Akka 进行配置，使之能够创建集群。我们需要对项目做一些配置：首先，把 Cluster 加入项目依赖；接着在 application.conf 中加入合适的配置项。在我们介绍项目中所有需要的配置时，读者可以使用 activator new 来新建一个项目。可以从 GitHub 上找到完整的项目：https://github.com/jasongoodwin/ learning-akka/tree/master/ch6，同时包含 Java 和 Scala 的实现。

在 build.sbt 文件中，首先需要加入 Akka Cluster 的依赖：

```
"com.typesafe.akka" %% "akka-cluster" % "2.3.6"
```

还需要加入 contrib 包。这个模块包含了非 Akka 团队对 Akka 贡献的一些功能，其中有一个 Akka Cluster 客户端，使用起来更加简单。之后我们将学习到客户端的功能。

```
"com.typesafe.akka" %% "akka-contrib" % "2.3.6"
```

既然已经把 Akka Cluster 添加到了项目中，我们就需要在 src/main/resources/application.conf 中添加集群配置。

```
akka {
    actor {
        provider = "akka.cluster.ClusterActorRefProvider"
    }
    remote {
        netty.tcp {
            hostname = "127.0.0.1"
            port = 2552 }
        }
    }
    cluster {
        seed-nodes = [
            "akka.tcp://Akkademy@127.0.0.1:2552",
            "akka.tcp://Akkademy@127.0.0.1:2551"]
    }
    extensions = ["akka.contrib.pattern.ClusterReceptionistExtension"]
}
```

下面是一些重要点。

首先，Cluster 的配置和 Remoting 的配置非常像，但是我们把 provider 改成了 ClusterActorRefProvider。

我们要指定主机和端口。在这个例子中使用了 Akka 默认的 2552 端口，并且指定本

地主机的 IP 用于测试。要在单台机器上测试一个集群，就需要在不同的端口上启动实例。我们可以通过在命令行内向 sbt 传递参数来完成这一点。如果需要传入任何参数，可以使用下面的命令：

```
activator run -Dakka.remote.netty.tcp.port=0
```

传入 port=0 表示由 Akka 随机分配一个端口。

我们还指定了种子节点。下文会介绍种子节点的具体定义，不过要注意主机、端口以及 ActorSystem 就是在配置种子节点时指定的。一定要确保指定的 ActorSystem 就是用于希望加入的集群。由于同一个实例内可以运行多个 ActorSystem，因此单单指定主机和端口并不足以确保成功连接。

最后一行指定了需要的扩展，我们在这行中加入了对于 contrib 包中 cluster 客户端的支持，稍后我们将详细介绍。

种子节点

读者可能很好奇 akka.cluster.seed-nodes 这个配置的作用。由于集群中可以包含任意数量的节点，我们可能不知道所有节点的地址。尤其是部署在云端的时候，部署的拓扑和 IP 地址可能经常变化。

还好我们有 gossip 协议，只要知道几个节点的位置就可以解决这个问题了。大多数技术（比如 Cassandra 和 Akka）把这些节点称作种子节点。除了知道它们的访问地址外，种子节点并无其他特殊之处。

为了理解种子节点的具体机制，我们将介绍节点加入集群的方式。当一个新节点加入集群时，该节点会尝试连接第一个种子节点。如果成功连接种子节点，新节点就会发布其地址（主机和端口）。种子节点会负责通过 gossip 协议将新节点的地址最终通知整个集群。如果连接第一个种子节点失败，新节点就会尝试连接第二个种子节点。只要成功连接任何一个种子节点，那么任何节点加入或离开集群时，我们都不需要对配置进行任何修改。

当部署到生产环境时，应该至少定义两个拥有固定 IP 的种子节点，并且保证任何时候都至少有一个种子节点可用。当一个节点尝试加入集群时，会试图顺序连接种子节点。如果所有的种子节点都不可用，那么新节点将无法加入到集群，如图 6-5 所示。

注意

在启动集群的种子节点时，对启动的顺序并没有要求。但是必须要启动列出的第一个种子节点，对集群进行初始化。

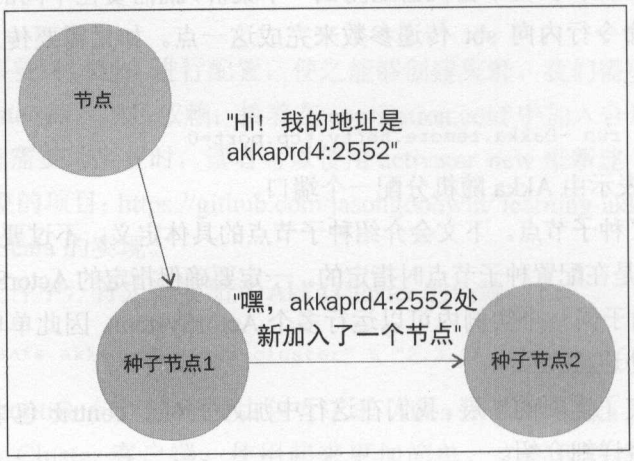


图 6-5

订阅集群事件

现在已经完成了在运行时创建一个集群所需的全部配置，所以可以开始编写代码了。我们将订阅集群事件，将集群环形拓扑中发生的所有变化都记录到日志。代码编写完成后，我们会对代码进行测试，然后继续学习如何在此基础上设计分布式服务和数据库。

首先，创建一个 Actor，命名为 ClusterController，这个 Actor 会作为其他例子的基础。接着，编写代码，使之在事件发生时执行特定操作。我们将首先创建 Actor，实例化一个 Logger，接着创建集群对象。

在 Java 中，Actor 的代码如下所示：

```
import static.akka.cluster.ClusterEvent.*;
public class ClusterController extends AbstractActor {
    protected final LoggingAdapter log = Logging.getLogger(
        context().system(), this);
    Cluster cluster = Cluster.get(getContext().system());
    @Override
    public void preStart() {
        cluster.subscribe(self(), initialStateAsEvents(),
            MemberEvent.class, UnreachableMember.class);
    }
    @Override
    public void postStop() {
        cluster.unsubscribe(self());
    }
    private ClusterController() {
        receive(ReceiveBuilder
```

```
                .match(MemberEvent.class, message -> {
                    log.info("MemberEvent: {}", message);
                })
                .match(UnreachableMember.class, message -> {
                    log.info("UnreachableMember: {}", message);
                )
                .build()
        );
    }
}
```

在 Scala 中，Actor 的代码如下所示：

```
class ClusterController extends Actor {
  val log = Logging(context.system, this)
  val cluster = Cluster(context.system)
  override def preStart() {
    cluster.subscribe(self, classOf[MemberEvent],
      classOf[UnreachableMember])
  }
  override def postStop() {
    cluster.unsubscribe(self)
  }
  override def receive = {
    case x: MemberEvent => log.info("MemberEvent: {}", x)
    case x: UnreachableMember =>
      log.info("UnreachableMember {}: ", x)
  }
}
```

首先定义一个 Logger。然后获取指向 Cluster 对象的引用。对于 Cluster 对象及其可用方法的介绍会贯穿本章。

使用 Actor 的 preStart 和 postStop 方法来订阅感兴趣的事件，要记得在 postStop 方法中调用 unsubscribe，防止泄露。Actor 将订阅下面两个事件：

- **MemberEvent**：该事件会在集群状态发生变化时发出通知；
- **UnreachableMember**：该事件会在某个节点被标记为不可用时发出通知。

接着，定义 Actor 在接收到上述事件时的行为：暂时直接输出到日志。稍后会介绍集群中发生的不同事件。

启动集群

现在将启动几个节点，确认所有配置都准确无误。首先，我们需要一个 main 函数，作为以程序方式启动 Actor 系统的入口。然后，创建 ClusterController Actor。

Java：

```
public class Main {
    public static void main(String... args) {
        ActorSystem system = ActorSystem.create("Akkademy");
        ActorRef clusterController = system.actorOf(
            Props.create(ClusterController.class),
            "clusterController");
    }
}
```

Scala：

```
object Main extends App {
  val system = ActorSystem("Akkademy")
  val clusterController = system.actorOf(
    Props[ClusterController], "clusterController")
}
```

直接运行下面的命令就可以启动种子节点：

```
activator run
```

不过为了节点能够安全退出集群，我们还将启用节点的 jmx 远程原理功能。所以我们还会在上面的命令中再传入几个参数：指定节点的 jmx 端口，并且在测试中关闭 jmx 的安全功能。稍后我们将介绍启用 jmx 远程管理功能的原因。加上 jmx 配置后，启动节点的命令如下：

```
activator run \
-Dcom.sun.management.jmxremote.port=9552 \
-Dcom.sun.management.jmxremote.authenticate=false \
-Dcom.sun.management.jmxremote.ssl=false
```

该命令会在配置好的 2552 端口上启动节点。我们将看到日志中输出了一些事件，表示第一个种子节点已经启动完成。也有可能由于节点尝试连接其他配置好的种子节点而看到几条 dead-letters 消息。最终，我们将看到接收到 MemberEvent 时输出到日志的内容：

```
[INFO] [06/14/2015 12:22:46.756] [Akkademy-akka.actor.default-
dispatcher-3] [akka://Akkademy/user/clusterController] MemberEvent:
MemberUp(Member(address = akka.tcp://Akkademy@127.0.0.1:2552, status =
Up))
```

在另一个终端窗口中，传入一个 Java 参数指定端口，在 2551 端口上启动第二个种子节点：

```
activator run \
-Dakka.remote.netty.tcp.port=2551 \
-Dcom.sun.management.jmxremote.port=9551 \
-Dcom.sun.management.jmxremote.authenticate=false \
-Dcom.sun.management.jmxremote.ssl=false
```

运行在 2552 端口的第一个种子节点会在第二个种子节点连接成功时将状态变化输出到日志：

```
[INFO] [06/14/2015 12:24:40.745] [Akkademy-akka.actor.default-
dispatcher-18] [akka://Akkademy/user/clusterController] MemberEvent:
MemberUp(Member(address = akka.tcp://Akkademy@127.0.0.1:2551, status =
Up))
```

现在我们就再添加一个节点，展示非种子节点的配置。我们给种子节点指定了端口，现在就将端口指定为 0，由 Akka 来随机分配一个端口。

```
activator run -Dakka.remote.netty.tcp.port=0 \
 -Dcom.sun.management.jmxremote.port=9553 \
 -Dcom.sun.management.jmxremote.authenticate=false \
 -Dcom.sun.management.jmxremote.ssl=false
```

稍微过一段时间后，就可以看到第三个节点连接到了集群。我们现在就有了构建分布式 Actor 系统的基础了。其中的很多步骤如果要自己实现的话还是有点困难的，但是 Akka 帮我们解决了这些问题。

注意

再次提醒，要记得启动种子节点列表中的第一个节点，这样集群才能完成初始化。

优雅地退出集群

如果试图通过杀死（kill）进程来关闭某个节点的话，就会发现 Akka 会把这个节点标记为不可到达，然后输出一些错误信息。在这种情况下，关闭节点会导致该节点无法访问，Akka 最终会将其标记为关闭。原因在于我们并没有优雅地退出集群。在将一个节点从集群中删除之前，我们应该通知集群：我们要移除这个节点了。

要在程序中完成这一操作，我们可以调用 cluster.leave，传入想要删除的节点的地址：

```
cluster.leave(self().path().address());
```

不过目前我们还没有编写任何 API，向外提供这一功能。所以我们将使用 jmx 和命令行工具来提供优雅删除节点的功能。我们将使用 Akka 发行版中附带的一个工具：akka-cluster。

我们可能需要先下载 Akka 发行版来获取 akka-cluster。可以从 http://akka.io/downloads/下载。将文件解压缩至 bin 文件夹。

现在，就可以使用这个工具箱集群发送命令了。最主要的用处就是优雅地从集群中删除节点。可以使用下面的方法，关闭端口 2552 上的种子节点（该节点暴露了 9552 端口用于 JMX）：

```
./akka-cluster localhost 9552 leave akka.tcp://Akkademy@127.0.0.1:2552
```

执行该操作后，我们可以看到该节点的状态从 Up 变为 Exiting，最后变为 Removed。

```
[INFO] [06/15/2015 20:05:21.501] [Akkademy-akka.actor.default-
dispatcher-3] [akka://Akkademy/user/clusterController] MemberEvent:
MemberExited(Member(address = akka.tcp://Akkademy@127.0.0.1:2552,
status = Exiting))
 [INFO] [06/15/2015 20:05:26.470] [Akkademy-akka.actor.default-
dispatcher-17] [akka://Akkademy/user/clusterController] MemberEvent:
MemberRemoved(Member(address = akka.tcp://Akkademy@127.0.0.1:2552,
status = Removed),Exiting)
```

6.5.2　集群成员的状态

加入集群后的节点可能会处在多种状态中的一种。在底层有一个逻辑上的 leader 节点，负责协调状态的变化。集群会从逻辑上对节点进行排序，集群中的所有节点都会遵循这个顺序。排序列表中的第一个节点就是 leader 节点。

Leader 节点会对加入和离开集群的请求做出响应，并修改集群成员的状态。

在加入集群时，将要加入的节点会将其状态设为 Joining。Leader 节点会做出响应，将其状态改为 Up。同样地，如果一个节点将状态设置为 Leaving，那么 leader 节点会做出响应，先将其状态改为 Exiting，然后再改成 Removed。所有这些状态改变都是通过 MemberEvent 在集群间发送的，我们订阅的就是 MemberEvent，并在发生该事件时输出日志。

失败检测

在 Actor 中还可以处理集群中可能出现的另一种情况：集群中的节点可以通过失败检测，检测其他节点是否可以到达。当某个节点出于任何原因被检测为不可到达时（比

如发生崩溃或是暂时的网络错误），该节点的状态不会发生变化，但是会被标记为 MemberUnreachable。在 ClusterController 中订阅这个事件，这样每次标记 MemberUnreachable 的时候就能收到通知。如果在合理的时间段内该节点又重新变得可达，那么该节点就会重新运行。如果在配置的时间内始终不可到达，那么 leader 节点就会将该节点标记为 Down，该节点将无法重新加入集群。

从功能性的角度来看，也可以直接监控成员状态的变化，但是失败检测的实现其实是基于从集群中收集到的数据计算出的不可达可能性（phi）。如果读者想要进一步了解的话，Cluster Specification 中有一个失败检测实现方法的链接。

我们将使用默认配置，但是在实际部署中，一定要阅读文档，根据网络可靠性调整配置。Amazon EC2 Instance（AWS）的可靠性明显要比小型的局域网络差很多，所以和自己的虚拟设施及网络设备比起来，使用 AWS 的时候可能需要对发生临时分区访问故障时的响应速度做更多考虑。在云环境中进行大规模部署后，我发现由于临时网络故障导致的服务中断要比普通操作导致的异常更为常见。

值得注意的是，如果一个节点被标记为不可达，那么 Akka 集群不会改变任何节点的状态：也就是说，在该节点由于无法从不可达状态恢复，被标记为 Down 之前，Akka Cluster 将不会改变任何节点的状态。

如果节点无法访问并且被标记为 Down，那么该节点在这之后将服务重新加入集群。结果会产生两个分离的集群（形成所谓的“左右脑”的情况）。目前 Akka 没有解决这个问题。所以一旦某个节点被标记为 Down，就必须关闭该节点，重新启动，获得一个新的唯一 ID，才能重新加入集群。

Akka Cluster 的所有基本知识就都介绍完了。接下来我们将学习如何在我们的集群上编写一些例子。

6.5.3 通过路由向集群发送消息

我们已经学习了如何创建集群。现在就来学习如何向集群发送消息。在这里将再次使用解析文章的问题，并且分配一个节点集群来完成这一任务。

6.5.4 编写分布式文章解析服务

第一个例子是编写一个可以水平扩展的分布式服务。我们将编写一个包含了文章解析服务的集群，然后通过一个客户端，将消息路由发送给集群中的任意成员。

在“第 5 章：纵向扩展”中，已经学习了如何使用 Actor Pool 从网页解析文章。我们将重用所有的代码，然后在示例集群中运行。集群中 Actor 的 Receive 块如下所示：

```
//Java
match(ParseArticle.class, x -> {
    ArticleParser.apply(x.htmlBody)
    .onSuccess(body -> sender().tell(body, self()))
    .onFailure(t -> sender().tell(new Status.Failure(t), self()));
})
//Scala
override def receive: Receive = {
  case ParseArticle(htmlString) =>
    val body: String = ArticleParser(htmlString)
    sender() ! body
}
```

读者可以在 GitHub 上查看“第 5 章：纵向扩展”的完整示例。我们会把 Pool 中用于解析文章的 Actor 放到集群上，然后启动集群。启动完成后，我们就将介绍如何从另一个 Actor 系统访问集群中的服务。要把这些服务放到集群上，只需要在集群项目里添加所需的依赖，然后把 ArticleParser，ArticleParseActor 和 ParseArticle 消息添加到应用程序中。

完成了上面的步骤后，就可以像本节中已经介绍过的一样，在启动集群时直接启动 Actor（或是 Pool 中的多个 Actor）。Main 函数如下所示：

```
public class Main {
    public static void main(String... args) {
        ActorSystem system = ActorSystem.create("Akkademy");
        ActorRef clusterController = system.actorOf(
            Props.create(ClusterController.class), "clusterController");
        ActorRef workerPool = system.actorOf(
            new BalancingPool(5).props(
                Props.create(ArticleParseActor.class)), "workers");
    }
}
```

本小节之前已经介绍过，可以在启动集群的时候启动一些节点。为了简单起见，我们将直接启动，不启用远程管理。如果使用的是 Linux，可以把启动的任务放到后台运行：

```
activator run &
activator run -Dakka.remote.netty.tcp.port=2551 &
activator run -Dakka.remote.netty.tcp.port=0 &
```

现在我们就启动了一个包含 3 个节点的集群。接下来将介绍客户端。

6.5.5 用于集群服务的集群客户端

编写一个客户端用于与无状态集群服务进行通信是相当直接的。和使用前置负载均衡器的传统 Web 服务相比，Akka Cluster 提供了一些便利之处：集群可以动态扩展或收缩，无需修改负载均衡器的配置。因为客户端可以将消息路由至集群中的任意成员，所以对基础设施的要求就更低一些。由于客户端知道集群信息，所以当集群内包含的节点数增加或减少时，客户端也会修改其可以发送消息的服务列表。客户端内部通过负载均衡将请求发送到集群中所有节点的原理如图 6-6 所示。

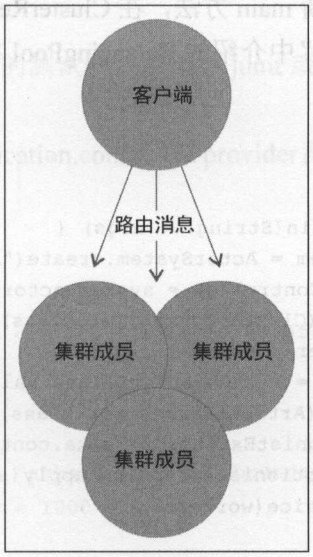


图 6-6

我们运行了一个包含 3 个节点的集群，下面就是客户端与其通信的必需条件：

1. 在服务器项目中启用集群客户端。
2. 客户端中必须包含要发送给服务的消息。
3. 客户端本身不是集群成员，但是必需知道集群的拓扑结构。所以我们将使用 contrib 库中的 Akka Cluster Client。
4. 客户端必须知道要将消息发送到哪些 Actor 或 Router，以及由哪些 Actor 或 Router 来响应集群事件。

设置服务器项目

在服务器项目中，我们要添加使用 Cluster Client 必需的依赖。Cluster Client 在 contrib

包中，这个包中包含了非 Akka 团队贡献的代码。添加下述依赖：

```
"com.typesafe.akka" %% "akka-contrib" % "2.3.6"
```

接着，在配置文件（application.conf）中为客户端添加 Akka 扩展。

```
akka.extensions = ["akka.contrib.pattern.
ClusterReceptionistExtension"]
```

这样就会在服务器上启动 ClusterReceptionist，负责处理客户端与集群之间的通信细节。服务器上创建 ClusterReceptionistActor 的路径是/user/receptionist。稍后我们将对此进行介绍。

完成上述步骤后，需要更新 main 方法，在 ClusterReceptionist 中注册 worker Actor。我们将使用“第 5 章：纵向扩展”中介绍的 BalancingPool 在每台服务器上创建一个 Actor Pool。

首先来看一下 Java 代码：

```
public class Main {
    public static void main(String... args) {
        ActorSystem system = ActorSystem.create("Akkademy");
        ActorRef clusterController = system.actorOf(
            Props.create(ClusterController.class), "clusterController");
        // router at /user/workers
        ActorRef workers = system.actorOf(new BalancingPool(5).props(
            Props.create(ArticleParseActor.class)), "workers");
        ((ClusterReceptionistExtension) akka.contrib.pattern
            .ClusterReceptionistExtension.apply(system))
            .registerService(workers);
    }
}
```

可以看到，我们调用的是 Scala API 的 apply 方法，然后需要强制类型转换至 ClusterReceptionistExtension。这完全适用于 worker Actor 的注册。

使用 Scala 的例子看上去更简单一些：

```
object Main extends App {
  val system = ActorSystem("Akkademy")
  val clusterController = system.actorOf(
    Props[ClusterController], "clusterController")
  // router at /user/workers
  val workers =
    system.actorOf(BalancingPool(5).props(
      Props[ArticleParseActor]), "workers")
  ClusterReceptionistExtension(system).registerService(workers)
}
```

如果之前已经运行了一些节点，那么我们需要在做了上述修改后重启这些节点。

设置客户端项目

可以为客户端新建一个项目。运行 activator new，选择想使用的语言模板。

编辑新项目的 build.sbt 文件。需要添加下述依赖：

```
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor" % "2.3.6",
  "com.typesafe.akka" %% "akka-cluster" % "2.3.6",
  "com.typesafe.akka" %% "akka-contrib" % "2.3.6"
)
```

读者还可以添加任何喜欢的测试依赖，比如 junit 或是 scalatest。对于上面的依赖我们应该都不陌生。

接着需要新建并编辑 application.conf，确保 provider 配置正确，并且设置 Cluster Client 的邮箱。

```
akka {
  actor {
    provider = "akka.cluster.ClusterActorRefProvider"
  }
contrib.cluster.client {
    mailbox {
      mailbox-type = "akka.dispatch.UnboundedDequeBasedMailbox"
      stash-capacity = 1000
    }
  }
}
```

在连接成功之前，请注意不要往邮箱发送太多消息。Cluster Client 会暂存这些消息。

在客户端和服务器之间共享消息

有多种不同的方法可以用来在客户端和服务器应用程序之间共享消息。我们可以在两个项目中都包含消息定义，也可以在服务器项目中添加客户端依赖（反之亦可），还可以在创建一个单独的项目用于消息定义，在服务器和客户端项目中都添加对该项目的依赖。在“第 2 章：Actor 与并发”中我们介绍过这一点，并且把包含消息定义的库发布到了本地仓库。可以安装 Nexus 或 Artifactory 作为团队闭源项目中发布仓库。对于开源项目，可以使用 Sonatype 的 OSS 仓库和 Maven Central。

如果一直跟着本书进度的话，可以选择任何一种方法：推荐在两个项目中都包含消息定义。为了简单起见，我们将在例子中修改 ArticleParseActor，接收并返回字符串：

```
//Java
match(String.class, x -> {
    ArticleParser.apply(x)
    .onSuccess(body -> sender().tell(body, self()))
    .onFailure(t -> sender().tell(new Status.Failure(t), self()));
//Scala
  case htmlString: String =>
    val body: String = ArticleParser(htmlString)
    sender() ! body
```

注意

对消息的修改可能会导致应用程序出错。在 Akka 中，可以将 Google Protocol Buffers 用于序列化，读者以后可能会想学习如何使用 Google Protocol Buffers 来管理消息的修改。

向集群发送消息

既然我们已经配置好了集群，可以和使用 contrib 包中 Cluster Client 的客户端进行通信，现在就可以向集群发送消息了。

我们将随机使用互联网上的一篇文章，抓取 HTML 源码，并创建一个叫做 articleToParse 的字符串变量。

把这个字符串发送给集群中的任意一个成员，应该从这个服务得到文章内容体。然后打印结果。如果发送了多次，可能就会发现集群中的所有成员都接收到了这条消息。

下面是客户端项目的 Java 代码：

```
public static void main(String[] args) throws Exception {
    Timeout timeout = new Timeout(Duration.create(5, "seconds"));
    ActorSystem system = ActorSystem.create("clientSystem");
    Set<ActorSelection> initialContacts = new HashSet<>();
    initialContacts.add(system.actorSelection(
        "akka.tcp://Akkademy@127.0.0.1:2552/user/receptionist"));
    initialContacts.add(system.actorSelection(
        "akka.tcp://Akkademy@127.0.0.1:2551/user/receptionist"));

    ActorRef receptionist = system.actorOf(
        ClusterClient.defaultProps(initialContacts));
```

```
    ClusterClient.Sendmsg = new ClusterClient.Send(
        "/user/workers", articleToParse, false);

    Future f = Patterns.ask(receptionist, msg, timeout);
    String result = (String) Await.result(f, timeout.duration());
    System.out.println("result: " + result);
}
```

下面是客户端项目的 Scala 代码：

```
def main(args: Array[String]) {
  val timeout = new Timeout(Duration.create(5, "seconds"))
  val system = ActorSystem.create("clientSystem")
  val initialContacts: Set[ActorSelection] = Set(
    system.actorSelection(
      "akka.tcp://Akkademy@127.0.0.1:2552/user/receptionist"),
    system.actorSelection(
      "akka.tcp://Akkademy@127.0.0.1:2551/user/receptionist")
  )
  import collection.JavaConversions._
  val receptionist = system.actorOf(
    ClusterClient.defaultProps(initialContacts))

  val msg = ClusterClient.Send("/user/workers", articleToParse, false)

  val f = Patterns.ask(receptionist, msg, timeout)
  val result = Await.result(f, timeout.duration).asInstanceOf[String]
  println("result: " + result)
}
```

下面我们一步一步地来解释代码。

首先，由于使用了 ask 并且要等待结果，因此在这里的测试中定义一个 timeout 变量。在实际情况中，可能会使用真正的 Actor 来发送并接收消息，这里只是一个例子。创建用于客户端的 ActorSystem。

现在，需要生成用于和集群中 Receptionist 进行通信的客户端。首先创建将会与客户端进行通信的包含种子节点的 Receptionist 清单。创建了包含种子节点的地址的 Set 后，我们就可以创建 ClusterClient Actor：

```
system.actorOf(ClusterClient.defaultProps(initialContacts))
```

现在，应用程序就可以连接到集群，并获取任何集群拓扑结构的更改信息了。远程 Actor 系统中的 Receptionist 可以接收一些不同的消息：

- ClusterClient.Send：将消息发送至任意一个节点。
- ClusterClient.SendToAll：将消息发送至集群中的所有节点。
- ClusterClient.Publish：将消息发送给订阅了某个主题的所有 Actor。

只需要将消息发送给任意一个节点，所以对于这种情况用 Send 就可以了。我们构建 Send 对象，指定消息要发送至的目标 Actor（运行在集群中每个节点上的 Router），并且传入要发送的消息：

```
new ClusterClient.Send("/user/workers", articleToParse, false)
```

最后，使用 ask 将消息发送至表示 Receptionist 的 ActorRef，然后得到返回的响应（上面调用了 ask 方法，不过在 Scala 中也可以使用?操作符）：

```
Patterns.ask(receptionist, msg, timeout);
```

Ask 会返回一个 Future，我们会等待 Future 返回结果。同样地，这只是用于例子。在真正的代码中不应该等待结果，阻塞线程。

注意

如果试图发送消息的时候节点变为不可用，请求就会超时并且失败。客户端应该负责超时超时和重试的语义或是使用其他方法处理错误。

要使用 Akka 来构建分布式 worker 节点，仅此而已。现在我们可以做很多事情来改进这个系统。如果可能会有非常多的消息，那么可能会想把消息放在其他地方，而不是内存中的信箱里。虽然对于实时处理来说，先使用临时内存可行，但是我们可能会要使用某种持久化队列或者是数据库。对于客户端来说，我们也可能需要编写超时和重试的语义，确保能够完成需要完成的任务。

构建分布式键值存储

上文已经介绍了基于 Akka Cluster 编写程序的所有内容，也介绍了一个使用无状态集群节点的小例子。如果涉及到状态的话，要正确解决问题就变得很困难了。

下一小节介绍处理包含状态的分布式系统（比如键值存储）的注意点、工具以及技术。

免责声明：分布式系统很难

在继续介绍如何构建分布式键值存储前，笔者提醒读者注意一些问题。

要编写看上去完美无误的分布式系统并不十分困难。你可能会很自信，觉得完全没那么难。可能没过多久，你就会觉得自己是分布式系统的专家了。但是请保持谦虚——实际上，失败情况经常发生，网络会发生分区访问故障，服务也可能会变得不可用，而要合适地处理这些情况，又不发生数据丢失和损坏是极为困难的。这个问题甚至是现在的网络技术无法解决的。

由于错误很常见，而且在程序扩展的过程中必然会发生，因此相对于应用程序的主要功能，如何在程序中响应上面这些错误情况甚至更为重要。

注意

要牢记，网络并不可靠。在一个运行在 AWS 上的包含 1000 个节点的集群中，任何时候系统中的某些服务都可能发生服务中断。

那么这些新兴的 NoSql 数据库实际上都是怎么实现他们声称的可用性和分区容错性的呢？在 aphyr.com 上有一个很有意思的系列文章，叫做 Jepsen 或是 Call Me Maybe，测试了一些分布式系统在其文档中关于可恢复性的声明。看到文章中对于一些我们最爱的技术的运行结果时，很可能会大吃一惊。

本书不可能解释用于解决这些问题的所有技术和解决方案，所以笔者给读者提醒：这件事非常不容易，而且从来都没有人完美地解决过分布式的问题。

6.5.6 集群设计

我们将构建一个包含三个节点的非常简单的数据库，以此展示可以用于分布式系统的技术。还将介绍几种不同的设计，帮助读者理解使用的技术以及分布式系统尝试解决的问题。

首先，先讨论一下集群，以及客户端如何与集群进行交互。理想情况下，我们需要包含多个节点的数据存储提供下述功能：

- 数据备份的机制，顺利地处理某个节点无法访问的情况；
- 提供备份一致性的机制——当用户请求数据时能获得最近更新的数据（一致性）；
- 线性扩展的机制——20 个节点的吞吐量是 10 个节点的两倍。

为了满足上面的需求，设计目标就是设计一个高可用、具备分区容错性、能够提供最新数据一致性要求的数据库。使用现在的技术要完美满足这三个需求是不太可能的，不过我们可以采取一些办法来尝试。幸运的是，已经有许多非常聪明的朋友研究过这些问题，因此我们可以借鉴他们的研究成果和发表的论文。

基本的键值存储设计

我们已经学习过如何在单个节点的 Map 中存储一个对象，其中每个对象都有一个 key。使用 HashMap 基本上可以提供常数时间的查询，所以是一种非常高效的在内存中存储数据的选择。

和“第 2 章：Actor 与并发”中完全一样，Actor 会接受包含 key 的 Get 消息以及包含 key 和 value 的 Put 消息。回顾一下，节点中会有一个如下所示的 Actor。

```
//Java
private AkkademyDb() {
    receive(ReceiveBuilder
        .match(SetRequest.class, message -> {
            log.info("Received Set request: {}", message);
            map.put(message.key, message.value);
            sender().tell(
                new Status.Success(message.key), self());
        })
        .match(GetRequest.class, message -> {
            log.info("Received Get request: {}", message);
            Object value = map.get(message.key);
            Object response = (value != null)
                ? value
                : new Status.Failure(
                    new KeyNotFoundException(message.key));
            sender().tell(response, self());
        })
        .matchAny(o ->
            sender().tell(
                new Status.Failure(new ClassNotFoundException()),
                self())
        )
        .build()
    );
}

//Scala
override def receive = {
  case SetRequest(key, value) =>
    log.info("received SetRequest - key: {} value: {}", key, value)
    map.put(key, value)
    sender() ! Status.Success
```

```
    case GetRequest(key) =>
      log.info("received GetRequest - key: {}", key)
      val response: Option[Object] = map.get(key)
      response match {
        case Some(x) => sender() ! x
        case None =>
          sender() ! Status.Failure(new KeyNotFoundException(key))
      }
    case o => Status.Failure(new ClassNotFoundException)
  }
```

如果试图实现这个 Actor，笔者也推荐添加其他用于常见用例的消息。比如类似 SetIfNotExist 的语义，在 key 已经存在的时候返回一个错误，在 key 完全不存在的时候设置 value。这对于处理并发一致性相当有用。如果想要查看对于一个需要处理多个客户端的键值存储来说比较重要的消息类型和数据类型的话，Redis API 是一个非常好的资源，文档简单易读，非常有助于我们理解这一问题。

节点间的协作

既然已经了解了在节点中存储数据的大概方法，现在就需要从更高一点的层次来了解如何在使用数据库的客户端和集群之间处理消息。

在客户端例子中，客户端会向任意一个节点发送消息，对于大多数分布式存储来说，这其实是个不错的开始。很多时候，这些系统会实现某种节点协作机制，集群中的任一节点接收到请求时，就会和集群中的其他节点进行通信协作，如图 6-7 所示。

读者可能现在对于这么做的原因还不是很清楚，不过可以先假设负责协作的节点需要跟另外 3 个节点通信才能够得到一个确定的值。

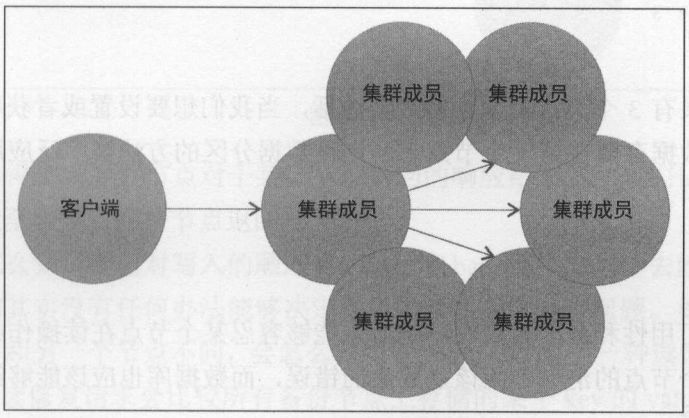


图 6-7

如果我们要实现这个逻辑，即一个客户端可以向集群中的任意节点发送请求，而接收到请求的节点就会作为该请求的协调者。该节点会向其他节点发送相应的请求，确定要返回给客户端请求的结果值。所以如果在服务器端而不是客户端来实现协调逻辑的话，可以继续使用随机发送机制。由于我们不知道客户端运行在哪里，所以把该逻辑放到集群中更为安全。在大多数情况下，我们对集群中节点的可用性更有信心一些。这也是 Cassandra 处理请求的方式。

我们将介绍两个模型：一个是把一份数据集存储在多个节点中；另一个是在多个节点中备份数据。

分区与线性扩展

我们要解决的第一个问题就是如何将某个问题域分配给集群中的不同节点。对于在键值存储中存储数据来说，要理解如何对数据分块（由 key 来决定）并分配给不同节点是比较容易的。在 Cassandra 中，用于决定要把数据分配给哪个节点的 key 叫做 partition key。

我们的例子是一个非常简单的键值存储，所以可以取一个 key 的哈希值，然后对哈希值取模，得到一个整数值。假设我们有 3 个节点用于存储数据。现在向集群中的某个节点发送请求存储一个键值对，比如 key 是 foo，value 是 bar。接收到请求的节点负责协调，执行 key 的 hashCode()方法：

```
"foo".hashCode()
 (101574)
```

然后对哈希值取模，得到 0，1，2 中的某个结果。

```
101574 % 3
 (0)
```

现在，如果有 3 个节点用于存储数据的话，当我们想要设置或者获取 key 为 foo 的值时，就知道数据存储在第 1 个节点上。这种数据分区的方法被广泛应用于现在的许多分布式数据库中。

节点冗余

要提供高可用性和分区容错性，就必须能够容忍某个节点在读操作或写操作时从集群中消失。某个节点的消失不应该是致命的错误，而数据库也应该能够继续运行，并且符合我们的需求。

要做到这一点，可以把要写入的数据发送给 3 个节点，并且期望能够收到大多数节点返回的确认。只有两个以上节点返回了成功，以后才可以读取此次写入的数据。我们现在无法介绍这个机制的所有细节，尤其是事件顺序的决定方式。如果读者对此有兴趣的话，可以了解一下 lamport 时钟或向量时钟。在这里，我们会使用更易于理解、更简单的机制从更高的层次加以介绍。

加入一个客户端想要写入一个键值对，key 为 foo，value 为 bar。我们希望将这个键值对存储到 3 个节点中。我们将尝试从负责协调的节点将数据写入 3 个节点。

我们可以参考前面的图表：客户端发起一个写入请求，集群中的 3 个节点都会收到该请求。要成功写入，就必须要有两个以上的节点确认写入成功。对最少成功写入节点数的修改会影响集群应对分区错误或节点故障的能力。

现在如果想读取数据，就可以向 3 个节点中的任意一个发送请求。但是如果某个节点停止运行了或是缺失某些写入的数据呢？在这种情况下，可以向所有 3 个节点都发送读取请求，只要至少两个节点返回相同的结果即可，如图 6-8 所示。

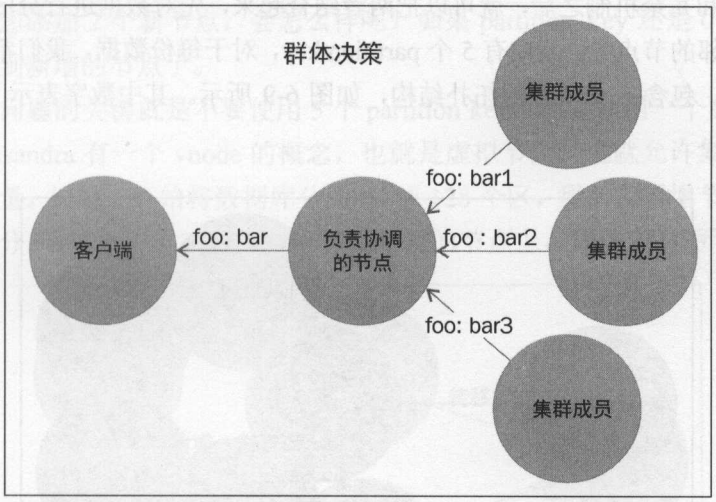


图 6-8

现在就能够得到大多数节点对于某个请求返回的响应结果了：我们认为要决定某个请求的结果，就至少要有两个节点返回相同的值。

但是我们怎么知道键值对写入的顺序呢？bar1 和 bar2 哪个先写进去的呢？哪个是最新的值呢？我们其实没有任何办法能够决定消息的顺序，这是个问题。如果某个节点上数据写入的顺序和另一个节点不同，会怎么样呢？Cassandra 有另一种读请求的类型，叫做读修复请求。读修复请求会比较所有备份节点上存储的某个 key 的 value，并且试图把最新写入的数据同步到所有的节点。

事件真正发生的时序是个很有意思的问题。我们可以先在记录中使用一个时间戳，确定哪个记录是最近写入的。但是无法保证所有的机器都使用相同的时钟，也无法保证时间戳的精确度能够区分高频发生的事件。

有很多论文和算法描述了这个问题并提供了解决方案。读者可作参考。其中比较有价值的是 Vector Clocks（Akka 和 Cassandra 等许多分布式技术中都使用了该技术）和 Dotted Version Vectors（Basho 的 Riak 最新版本使用了该技术）。

- 亚马逊发表的 Dynamo 的论文中描述了 Amazon Dynamo 分布式键值存储中对于 Vector Clocks 的使用。
- Lamport 在 1978 年写了一篇关于分布式系统时序问题的论文，也值得一看 -http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf

6.6　结合分区与冗余

有了分区和冗余机制之后，就可以把两者结合起来，先对数据进行分区，然后将数据备份存储到相邻的节点中。假如有 5 个 partition key，对于每份数据，我们希望在 3 个节点上保存其备份。包含 5 个节点的拓扑结构，如图 6-9 所示。其中数字表示了 partition key 的哈希值。

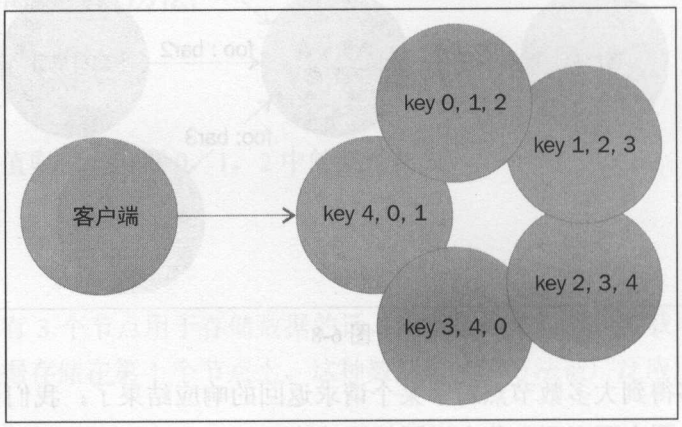


图 6-9

读取请求会发送至任意一个节点，该节点负责进行协调，计算 partition key 的哈希值，然后把请求发送给实际存储数据的 3 个节点。此时数据的读取和传递机制和已经介绍过的并无不同，如图 6-10 所示。

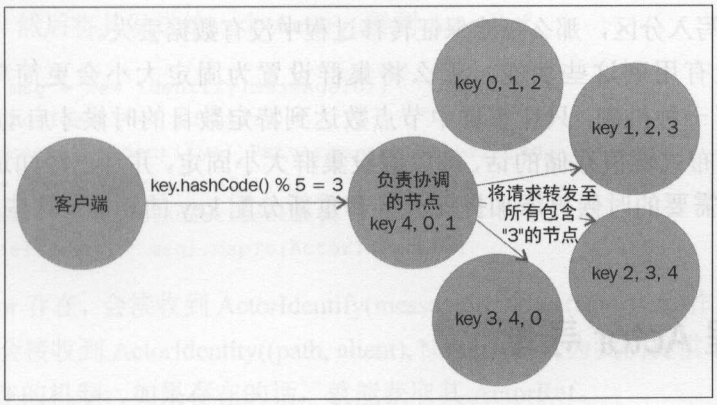


图 6-10

提前分区以及向新节点重新发送 key

如果希望能够添加新节点，会发生什么呢？例如，如果还是希望为每份数据保存 3 份备份，但是想添加 2 个新节点，会怎么样呢？如果 partition key 还是 0 到 4 的话，就无法把数据移到新增的节点了。

解决这个问题的关键就是不要使用 5 个 partition key，而是使用一个更大的 partition key 总数。Cassandra 有一个 vnode 的概念，也就是虚拟节点，这就允许集群中的分区数量大于节点数量。如果一开始将数据库分为 64 或 128 个区，那么在新增节点的时候就可以把存储一部分 partition key 对应的数据移到新增的节点上，如图 6-11 所示。

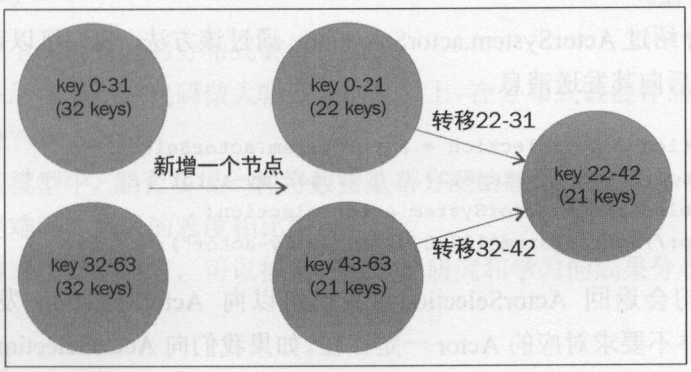


图 6-11

接着，负责协调的节点会和新加入的节点通信，告知新节点已经将一些 partition key 分配给它。要注意的是，向新节点转移数据是一个一次性的操作，可能需要停止所有节点上的所有操作，直到 partition key 的重新分配完成为止。如果在转移数据的过程中仍

然有大量数据写入分区，那么很难保证转移过程中没有数据丢失。

如果还没有用到这些功能，那么将集群设置为固定大小会更简单一些。Akka Cluster 提供了一种机制，只在集群中节点数达到特定数目的时候才启动集群。如果想要自己构建分布式键值存储的话，可以假设集群大小固定，并且已经知道 partition key 的数目，以后需要的时候再添加提前分区和重新分配 key 的功能（这些问题解决起来可都不简单）。

6.7　远程 Actor 寻址

上文深入介绍了实现细节，不过在本章的最后，我们要快速地再次介绍一下 Actor 的寻址，以及如何获取远程 Actor 的引用。

如果我们有一个 ActorRef 的话，可以调用 actorRef.path()得到 Actor 的 URL：akka://ActorSystem/user/actor。

Actor 的路径包含两部分：

- **源**：akka://ActorSystem；
- **路径**：/user/actor。

源可以在本地（akka://ActorSystem），也可以在远程（akka.tcp://Actor System@127.0.0.7）。无论是本地还是远程 Actor，路径的格式相同（例如/user/actor）。

ActorRef 除了包含 ActorPath 外还有 Actor 的 UID，比如#123456。ActorPath 仅仅包含了 Actor 的路径。

我们之前介绍过 ActorSystem.actorSelection。通过该方法，我们可以查询位于任意路径的 Actor，然后向其发送消息。

```
ActorSelection actorSelection = actorSystem.actorSelection(
    "akka.tcp://ActorSystem@127.0.0.7/user/my-actor");
val actorSelection = actorSystem.actorSelection(
  "akka.tcp://ActorSystem@127.0.0.7/user/my-actor")
```

上面的语句会返回 ActorSelection，我们可以向 ActorSelection 发送消息。不过 ActorSelection 并不要求对应的 Actor 一定存在。如果我们向 ActorSelection 发送消息而其对应的 Actor 不存在，消息就会丢失。因此我们可以查询远程 Actor，得到指向其的 ActorRef。

使用 akka.actor.Identify 寻找远程 Actor

所有 Actor 默认情况下都接受一条消息：akka.actor.Identify。我们可以新建一条

Identify 消息，然后将其发送给一个 Actor，得到 ActorRef。

```
Identify msg = new Identify(messageId);
Future<ActorIdentity> identity =
    (Future<ActorIdentity>) Patterns.ask(actor, msg, timeout);
val msg = Identify(messageId)
val identity: Future[ActorIdentity] =
  (sentinelClient ? msg).mapTo[ActorIdentity]
```

如果 Actor 存在，会接收到 ActorIdentify(messageId, Some(actorRef))作为响应；如果不存在的话，就会接收到 ActorIdentity((path, client), None)。这就为我们提供了一种判断远程 Actor 是否存在的机制，如果存在的话，就能获取其 ActorRef。

6.8 课后作业

本章结合理论与技术，介绍了如何使用 Akka Cluster 来执行任务，以及如何通过 key 来对数据集分区并分配给不同的节点。这些问题非常复杂，而本章只是对 Akka Cluster 的基本介绍，所以并未深入讨论。如果读者想要尝试解决本章涉及的一些问题，可以开始去学习如今这些问题的常见解决方案。

- 使用 Akka Cluster 构建自己的工作队列。
- 构建一个支持冗余备份的分布式键值存储。
- 试着解决线性扩展的问题——如何判断时序，能否“修复”出现错误的节点并恢复运行？
- 构建一个支持分区的分布式键值存储。
- 能否在尽可能不对代码做大幅改动的基础上，在分布式键值存储中结合使用分区和备份？
- 在分区模型中，能否实现一种将数据重新分配给新节点的方法，如果使用备份的话会更难吗，两者的难度相比如何？
- 如果编写了一个项目，可以把它开源，把研究和学习的成果分享给他人。

6.9 小结

本章介绍了设计不同的分布式系统时所需的一些基本知识。这包括了解决不同问题时可以使用的一些模型——不要觉得只有在处理数据的时候才可以使用分区技术，在解决许多实时系统的水平扩展问题时也可以使用类似的技术。

如果想要深入学习本章介绍的技术，应该继续学习，并且试着自己实现这些问题的解决方案。学习的最好方法就是教授和分享。所以可以试着成立一个分布式计算的俱乐部，或者收集一些同事关于这些技术的演讲，然后组织一下自己的想法，这有助于我们理解细节，形成一套自己对于现在如何解决这些类型的分布式计算问题的看法。

在下一章中，我们将讨论 Actor 负载较大时邮箱中会发生什么，以及如果调整邮箱的配置来处理这些情况。

第 7 章
处理邮箱问题

恭喜！现在已经完成所有困难的部分了。我们已经学习了 Akka，如何扩展，以及如何在各种情况下描述系统的行为。

在本章中，我们将介绍当负载达到 Actor 系统极限时会发生的情况，以及如何描述系统在这些情况下的行为。

首先，我们将介绍要解决的问题，然后再学习可以用来解决这些问题的不同方法。

7.1 搞垮最可能出问题的服务

继续使用前文的例子，假设有一个应用程序，可以解析文章，并且把文章内容体存储在键值存储中。解析出的文章会通过阅读应用程序显示在各种不同的设备上。

假设已经启动了应用程序，而且用户量不断攀升。一切都进展顺利。用户可以在设备上使用应用程序通过一个公共的 API 向解析服务发送请求，然后阅读文章。解析服务会检查数据库，如果请求的文章还没有解析过的话，就会对其进行解析，并存储到缓存。

文章解析及存储的流程，如图 7-1 所示。

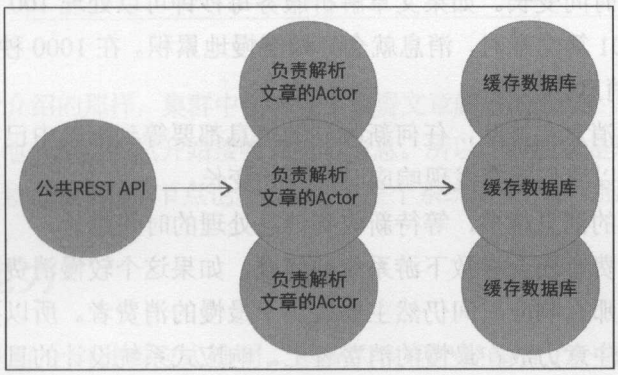


图 7-1

假如有一天发生了一些好事情。我们的应用程序在 Hacker News 上全天都被顶在了第一名，访问流量达到了历史最高值的 10 倍。

首先，请求 REST API 的时候会开始出现超时。响应时间大大增加。最终，文章解析服务由于内存溢出而崩溃，所以要开始分析应用程序的负载流量。由于文章解析服务是整个应用程序中对计算资源要求最多的部分，所以有许多消息都在队列中等候。这是整个数据流中最慢的一个服务，如图 7-2 所示。

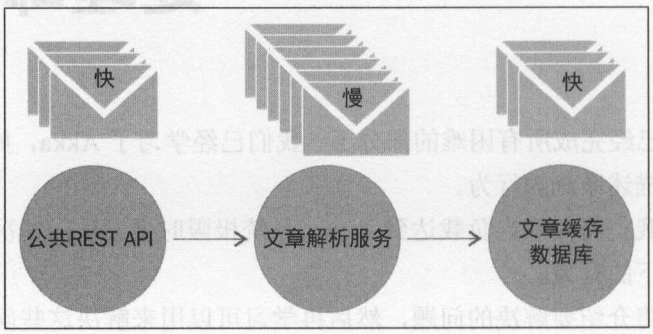


图 7-2

要知道消息会被发送到内存中的邮箱里。如果服务处理消息的速度比邮箱接收到消息的速度慢，那么邮箱中的消息会越来越多。我们来看一下发生上述事件时的现象，更深入地理解背后的原因。可以使用不同的配置应用程序的方法，确保下次发生类似情况时程序能够正常运行。

7.1.1　响应时间变长

当最慢的文章解析服务邮箱中的请求开始堆积得越来越多的时候，我们观察到的第一个现象就是响应时间变长。如果文章解析服务每秒钟可以处理 100 篇文章，那么当我们每秒向其发送 101 篇文章时，消息就会开始慢慢地累积。在 1000 秒之后，邮箱的队列中就会有 1000 条消息。

一旦队列中有消息在排队，任何新收到的消息都要等到邮箱中已有的消息都处理完成后才能处理，所以我们就会发现响应时间开始变长。

邮箱中未处理的消息越多，等待新收到消息处理的时间越长。

较慢的消息消费者还会导致下游系统的问题。如果这个较慢消费者的下游还有一个非常快的消费者，那么响应时间仍然主要取决于最慢的消费者。所以要降低系统的响应时间，就必须要把注意力放在最慢的消费者上。响应式系统设计的目标之一就是保证系统始终能够进行响应，所以这一现象违背了灵敏性的原则，如图 7-3 所示。

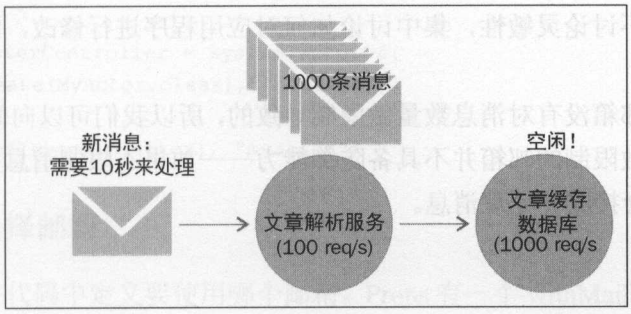


图 7-3

7.1.2 崩溃

如果继续向系统发送越来越多的消息，那么最终较慢的消费者的邮箱中会堆积大量消息，导致发生一个更严重的问题：由于内存溢出而崩溃。

一旦响应时间变得这么慢以后，想要使用服务的用户可能会不断地重试，向已经不堪重负的服务的邮箱中发送更多消息。

邮箱默认对消息数量没有限制，这意味着服务会无限累积消息。但是资源是有限的：JVM 只有这么多内存，所以最终较慢的消息消费者会在内存中保存大量消息，导致 JVM 内存不足，无法创建新对象，程序崩溃，如图 7-4 所示。

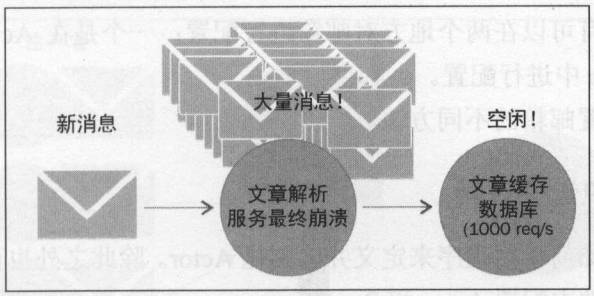


图 7-4

如果像上一章介绍的那样，集群中包含多个负责文章解析的节点，那么某个节点的崩溃意味着集群中的其他节点会突然开始接收到更多消息。所以影响通常是层层传播的——其他节点接收到更多消息，最终这些节点也会崩溃。当整个系统崩溃以后，服务就彻底不可用了。

7.2 恢复能力

前面的章节已经介绍过响应式准则。系统的表现已经违背了一些响应式准则：容错性和灵敏性。

我们现在先不讨论灵敏性，集中讨论如何对应用程序进行修改，防止其在负载突然增加时崩溃。

崩溃是由于邮箱没有对消息数量做限制导致的，所以我们可以向修改 Actor 的邮箱。没有对消息数量做限制的邮箱并不具备防御能力——如果不限制消息数量的话，我们认为应用程序永远会接收并处理消息。

7.2.1　邮箱

直到现在为止，都没有详细介绍过邮箱。我们只是知道消息存储在邮箱里，然后被处理。在服务被大规模使用之前，只要知道这些就绰绰有余了。这也是为什么我把本章放在全书接近结尾的地方——我们很早就介绍了邮箱，但是只有在处理实际网络流量的时候才会需要对其进行优化。

邮箱配置

在 Akka 中，有多种方法可以对邮箱进行配置。如果想了解更多细节的话，可以查阅 Akka Mailboxes 的文档。

几乎在任何情况下，Actor 都有自己的邮箱。唯一的例外是使用 Balancing Dispatcher 的 BalancingPool。我们在“第 5 章：纵向扩展”中提到过，BalancingPool 中的 Actor 共享同一个邮箱。所有可以在两个地方对邮箱进行配置：一个是在 Actor 里进行配置，另一个是在 Dispatcher 中进行配置。

我们将介绍配置邮箱的不同方法。

在部署配置中选择邮箱

我们已经知道如何使用程序来定义并实例化 Actor。除此之外也可以使用 Akka 的部署配置，在配置文件中配置 Actor 和 Router。

我们可以在部署配置中定义 Actor 的邮箱（通过 Actor 的路径）。部属配置中的定义在所有邮箱配置中优先级最高。可以在 application.conf 中对 Actor 邮箱定义如下。

```
akka.actor.deployment {
  /myactor {
    mailbox = default-mailbox
  }
}
```

这样创建于/user/myactor 的 Actor 就会使用默认邮箱：

```
ActorRef clusterController = system.actorOf(
    Props.create(MyActor.class), "myactor");

system.actorOf(Props[MyActor], "myactor")
```

在代码中选择邮箱

我们也可以在代码中定义要使用哪个邮箱。Props 有一个 withMailbox 方法，我们可以在创建 Actor 的时候调用该方法，为 Actor 分配邮箱：

```
ActorRef clusterController = system.actorOf(
    Props.create(MyActor.class).withMailbox("default-mailbox"));

system.actorOf(Props[MyActor].withMailbox("default-mailbox")
```

决定使用哪个邮箱

默认邮箱可以用于所有的用例，包括多个 Actor 之间会共享邮箱的 BalancingPool/ BalancingDispatcher。

除非使用 BalancingPool，否则邮箱中的消息只会有一个消费者，如图 7-5 所示。

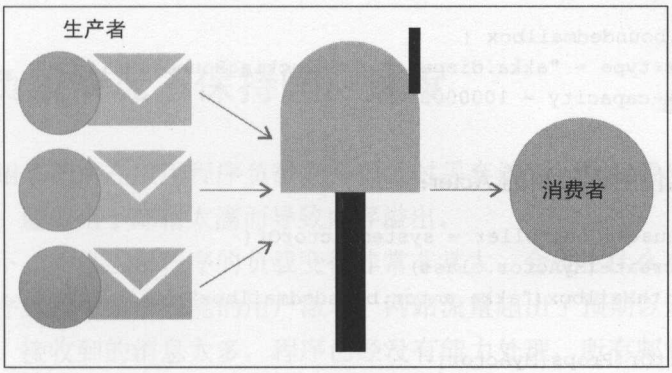


图 7-5

在单个消费者的情况下，除了默认邮箱外，还可以使用 SingleConsumerOnly UnboundedMailbox，该邮箱在大多数情况下效率优于默认邮箱（对于与效率有关的修改，一定要记得测试性能）。唯一不能使用该邮箱的情况就是 BalancingPool/ Balancing Dispatcher，原因在于会有多个 Actor 读取同一个邮箱中的消息。使用队列的实现只支持单个消费者。

首先在 application.conf 中定义一个邮箱：

```
akka.actor.mymailbox {
    mailbox-type = "akka.dispatch.SingleConsumerOnlyUnboundedMailbox"
}
```

然后创建使用该邮箱的 Actor：

```
ActorRef clusterController = system.actorOf(
    Props.create(MyActor.class)
        .withMailbox("akka.actor.mymailbox"));

system.actorOf(Props[MyActor].withMailbox("akka.actor.mymailbox")
```

我们经常会需要决定是否需要限制邮箱中消息的最大数目。在大多数情况，没有消息数目限制的邮箱效率更好，推荐使用。不过这个例子将使用有消息数目限制的邮箱，会在达到限制时拒绝接收消息。我们不希望在负载突然变大时出现内存溢出，反之，我们会拒绝接收消息。

有两种类型的限制消息数量的邮箱——阻塞的和非阻塞的。所有的邮箱都基于队列。在这种情况下，阻塞意味着当邮箱已满时，再向邮箱发送消息会导致线程等待直至邮箱腾出空间，而非阻塞意味着此时消息会被丢弃。由于我们已经决定丢弃消息，所以可以使用 NonBlockingBoundedMailbox。我们在配置文件中添加该邮箱：

```
akka.actor.boundedmailbox {
    mailbox-type = "akka.dispatch.NonBlockingBoundedMailbox"
    mailbox-capacity = 1000000
}
```

然后实例化使用该邮箱的 Actor：

```
ActorRef clusterController = system.actorOf(
    Props.create(MyActor.class)
        .withMailbox("akka.actor.boundedmailbox"));

system.actorOf(Props[MyActor]
    .withMailbox("akka.actor.boundedmailbox")
```

现在，如果系统负载突然变得很大，就将丢弃消息，但是系统仍将正常运行。

那么如果消息被丢弃，会发生什么呢？与下游系统进行通信的系统应该实现超时以及重试机制。我们要预设消息有时候可能会丢失，所以如果要保证消息至少能够被处理一次的话，构建系统的时候就需要记录请求的状态，无论任何原因导致请求失败，都要重发请求。在我们的示例应用程序中，客户端可能会发生错误，所以用户可以选择重试。而在我们自己的系统中，要实现超时和重试机制都很容易。

提高邮箱中消息的优先级

我们还应该了解另外两大类的邮箱：优先级邮箱和控制感知的邮箱。它们的目的相同：支持消息处理的顺序。

优先级邮箱在接收到消息之后，可以对消息进行排序，给每个消息赋予一个优先级。这会造成明显的额外性能开销：每次接收到消息时都要重新排序，这意味着消息队列的生产者和消费者都必须要等待消息重新排列完成。幸运的是，消息的重排序在大多数情况并不重要，所以大多数情况下我们都不需要使用这些邮箱。

不过有一种情况比较常见：如果需要给 Actor 提供某种控制消息，告诉 Actor 发生了某些改变，而这些改变会影响处理队列中消息的方式。

幸运的是，还有一种邮箱类型：ControlMessageAware 邮箱。该邮箱也通过一个高效队列来处理消息，但是允许将任意扩展了 akka.dispatch.Controlmessage 的消息插入到队列头部。如果 Actor 从邮箱队列的剩余消息中获取消息进行处理时不需要太复杂的时序逻辑，那么这种解决方案比优先级队列要好，应该优先选择使用。

这两种邮箱使用得很少，所以我们不需要深入了解，但是知道它们的存在还是很重要的。Akka 文档深入介绍了两者的使用。

最后值得一提的一个注意点是，如果所有邮箱都不符合我们的使用场景，那么可以创建自己的邮箱。

7.3 在高负载情况下保持响应速度

现在已经介绍了如何在应用程序负载变得很大时丢弃消息。这就确保了应用程序此时能够继续运行，避免由于邮箱太满而导致内存溢出。

我们来看一下，一旦应用程序的负载变得非常非常大，会发生什么事情。

假设应用程序过程中某个功能的用户激增，网站流量超出了预期以及应用的处理能力。和之前一样，接收到的消息太多，程序已经没有能力处理，所有邮箱中开始堆积消息。由于必须要先处理完邮箱中堆积的消息才能处理新接收到的消息，所以处理消息所需的时间就变长了。

最终，在持续的高负载下，邮箱中消息的数量到达了配置的最大值。此时，邮箱就会开始丢弃消息。而如果使用阻塞的有消息数量限制的邮箱，那么线程就会等待邮箱中有空间再继续执行。无论选用哪种方法，现在我们都能够确保应用程序不会崩溃。

但是这真的是个好方法么？我们可以从多个层次来考量对邮箱中消息数量的限制，以决定如何定义程序的行为：

- 如果将邮箱消息数量限制设置得较小，那么任何网络流量的激增都会导致消息丢失，这样可能可以保证应用程序能够在较短的时间内进行响应。
- 如果将邮箱消息数量限制设置得较大，请求就可能在 Actor 能够做出响应之前超时。

无论使用哪种方式，用户都需要等待超时的发生。如果邮箱非常小，被丢弃的消息就直接消失了，所以任何等待响应的用户就会看到请求超时并失败。如果邮箱非常大或是对消息数量没有限制，那么即使消息最终能够被处理，等待响应的客户端（比如用户的浏览器或发送请求的应用程序）最终还是会超时并失败。我们必需具体问题具体分析，衡量上面的两种方法是否适合。如果并不太关注响应，并且在流量激增时能够容忍消息丢失，那么可以采用这两种方法。

不过大多数情况下，要在实时系统中等待很长时间都是不太能够接受的，尤其是等待了很长时间后却只收到了错误信息作为响应。

7.3.1　熔断

当把灵敏性作为系统的目标时，任何使得用户等待很长时间后接收到错误的情况实际上都是不可接受的。同样地，如果把容错性作为目标，那么也就不应该允许系统的某个部件在消息数量激增时变得完全不可用。

熔断机制会监控应用程序某些部分的响应延迟或错误。消息会先照常发送给熔断器，熔断器负责监控消息的响应时间。此时状态为“关闭”，如图 7-6 所示。

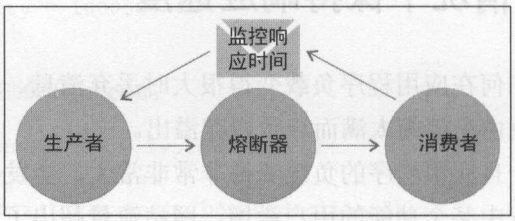


图 7-6

一旦达到了某个特定的延迟上限，熔断器就会把状态修改为“打开”，此时会立刻拒绝所有收到的请求，如图 7-7 所示。

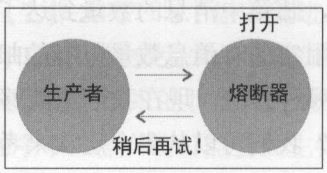


图 7-7

接着，在一段时间后，熔断器就会把状态改为“半开”，并尝试发送一条消息，如图 7-8 所示。

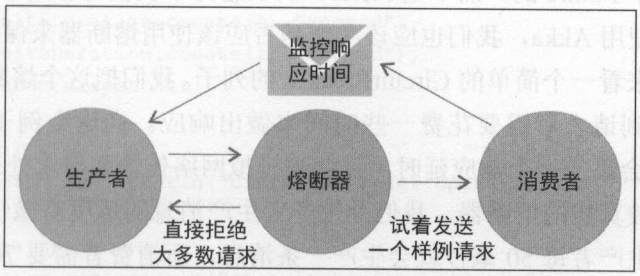


图 7-8

如果请求很快就得到响应，那么可以认为服务已经恢复，此时熔断器把状态修改为“关闭”（译者注：此处原文有误，已更正）。

事实证明这就是我们希望达到的效果。

如果熔断器处于“打开”状态，那么此时系统可以达到下述要求：

- **灵敏性**：如果发生错误，系统可以马上返回响应；
- **容错性**：熔断器有助于下游系统恢复，而不是不断向下游系统发送请求直至其崩溃。

要注意的是，熔断器对于错误和超时的响应方法是相同的。如果发生了许多错误，那么也会导致熔断器状态变为“打开”。Akka 的熔断器不会区分超时和其他类型的失败情况（而我们可能需要进行区分。），但是我们当然可以进行自定义，自己实现这个区分逻辑。如果下游系统由于负载较重而无法正常运行，那么一般来说都能做出一个假设：它需要时间来恢复。所以实现熔断模式通常来说是保护下游系统的一个好办法。

熔断监听器

在生产环境使用熔断器时，可能并不精确地知道熔断器何时以何种方式修改状态，所以我们需要做的就是收集数据。对于事物的运作方式经常做出错误的断言，所以要记住，一定要对做出的这些假设进行评测。

幸运的是，熔断器提供了一些可供重写的函数，使得我们可以在不同时间发生时添加一些行为：onOpen，onClose 以及 onHalfOpen。

要了解熔断器的实际运作过程，就应该这些事件发生时弹出提醒，或者至少是记录日志。

熔断示例

熔断器是基于 Future 的，而不是 Actor。所以它并不仅仅可以用于 Actor。甚至说如果没有在项目里使用 Akka，我们也应该考虑是否应该使用熔断器来保护系统。

现在我们就来看一个简单的 CircuitBreaker 的列子。我们把这个熔断器部署在服务前端，该服务接收到请求后需要花费一些时间来做出响应。在这个例子中，将使用本地 Actor，但是我们会添加一个响应延时，以此来模拟网络传输和请求处理的时间。

为了展示比较真实的熔断器，我们令生产者生产消息的速度略微快于目标 Actor 处理消息的速度。生产者每 50 毫秒就会生产一条消息，而消费者需要 70 毫秒才能完成对消息的响应。这样的话，队列中的消息就会慢慢积攒变多，响应时间也会越来越长，直到熔断器改变状态为止。

我们仍然使用键值存储的例子，并且设置 70 毫秒的延时。

Java 代码：

```
match(GetRequest.class, message -> {
    Thread.sleep(70); //slow down the actor's response
    Object value = map.get(message.key);
    Object response = (value != null)
        ? value
        : new Status.Failure(
            new KeyNotFoundException(message.key));
    sender().tell(response, self());
}
```

Scala 代码：

```
case GetRequest(key) =>
  Thread.sleep(70)
  val response: Option[Object] = map.get(key)
  response match {
    case Some(x) => sender() ! x
    case None =>
      sender() ! Status.Failure(new KeyNotFoundException(key))
  }
```

在发起调用的类中，我们将创建一个熔断器，当延时超过 1 秒时就会将状态改为“打开”。

Java 代码：

```
CircuitBreaker breaker =
    new CircuitBreaker(
        system.scheduler(),
        10,
        FiniteDuration.create(1, "second"),
        FiniteDuration.create(1, "second"),
        system.dispatcher()
    )
    .onOpen(() -> {
        System.out.println("circuit breaker opened!");
    })
    .onClose(() -> {
        System.out.println("circuit breaker closed!");
    })
    .onHalfOpen(() -> {
        System.out.println("circuit breaker half-open!");
    });
```

Scala 代码：

```
val breaker =
  new CircuitBreaker(
    system.scheduler,
    maxFailures = 10,
    callTimeout = 1 seconds,
    resetTimeout = 1 seconds
  )
  .onOpen(println("circuit breaker opened!"))
  .onClose(println("circuit breaker closed!"))
  .onHalfOpen(println("circuit breaker half-open"))
```

可以看到 Scala 代码中传入的参数和 Java 代码中完全相同。创建 CircuitBreaker 时，需要传入下述参数：

- 熔断器熔断之前发生错误的最大次数（无论是超时还是 Future 返回失败）；
- 调用超时（延时超过多长时间就熔断）；
- 重置超时（等待多久以后将状态改为“半开”，并尝试发送一个请求）。

接着，我们在熔断器将状态改为“打开”、“半开”以及“关闭”时注册日志事件。

熔断器的使用很简单：只要提供一个匿名函数，该匿名函数不需要参数，返回一个 Future。在这个例子中，我们将使用 ask。

我们每 50 毫秒发起一次调用，Actor 在 70 毫秒之后进行响应，所以队列中的消息将越来越多。当邮箱填满时，响应时间将变得非常长，最终熔断器会熔断。

Java 代码：

```
Timeout timeout = Timeout.apply(2000);

ActorRef db = system.actorOf(Props.create(SlowAkkademyDb.class));
Await.result(
    Patterns.ask(db, new SetRequest("key", "value"), timeout),
    timeout.duration());

for(int i = 0; i < 10000000; i++) {
    Future future = breaker.callWithCircuitBreaker(() ->
        Patterns.ask(db, new GetRequest("key"), timeout));
    toJava(future).handle((x,t) -> {
        if (t != null) {
            System.out.println("got it: " + x);
        } else {
            System.out.println("error: " + t.toString());
        }

        return null;
    });

    Thread.sleep(50);
}
```

Scala 代码：

```
implicit val timeout = Timeout(2 seconds)
val db = system.actorOf(Props[FastSlowAkkademyDb])
Await.result(db ? SetRequest("key", "value"), 2 seconds)

(1 to 1000000).map(x => {
  Thread.sleep(50)
  val askFuture = breaker.withCircuitBreaker(db ? GetRequest("key"))
  askFuture.map(x => "got it: " + x).recover({
    case t => "error: " + t.toString
  }).foreach(x => println(x))
})
```

在某些情况下，我们可能会想实现自己的熔断器，一定要注意 Future 何时会返回失败。熔断器对所有种类的失败情况一视同仁，只要达到最大失败次数，就会熔断。如果希望用和熔断器默认方式不同的方法来处理 Future 返回失败的情况，那么可以把响应封装在 Try 中，这样就可以令 Future 返回成功。

我们还可以考虑使用另一种策略来防止消费者接收到过多的请求。如果生产者会无限地发送请求，那么我们可以反转发送消息的方向。由 Actor 主动请求处理消息，而不是直接接收无穷无尽的消息。

另一个相关的概念是“反向压力（Back-Pressure）”，指的是减缓消息流的速度，使之与最慢的消费者的处理速度相当，这样就可以避免下游系统由于负载过高而瘫痪。我们当然可以自己实现类似的机制，不过 Reactive Streams 提案中定义了相关的标准来处理这个特定的问题。

Reactive Streams 的目标是提供一个使用非阻塞反向压力来进行异步流处理的标准。其试图提供的标准不止面向运行环境（JVM 和 JavaScript），同时也面向网络协议。

现在有好多个 Reactive Streams 的实现，包括 Akka 团队的实现（Akka Streams）。Reactive Streams 是一个相当大的话题，所以我们并不准备在这里介绍这项技术。不过当生产者几乎会无限发送请求时，我们要知道有人在积极地解决这个问题，并且特别值得关注一下 Reactive Streams。

Reactive Streams 混合使用了多种技术，在接收请求的时候如果服务负载过大就会减缓请求速度（反向压力），同时也会在有余力的时候主动请求额外的负载。

Reactive Streams 这个话题令人兴奋，但是在本书撰写时仍然处于起步阶段。Akka Streams 也仍然被标记为实验版本，不过离正式发布已经不远，而 Typesafe（译者注：已改名为 Lightbend）也花了许多时间，组织研讨会，让大家知道 Akka Streams 的进展。

读者可以从 http://www.reactive-streams.org/了解更多关于 Reactive Streams 的内容。

7.4 课后作业

尝试编写一个应用程序，发送消息填满邮箱。接着尝试不同的方法，防止邮箱被填满而导致的服务不可用。

- 尝试填满一个本地 Actor 系统的邮箱（可以令 Actor 的线程睡眠若干毫秒，模拟对请求的处理）。观察应用程序的运行情况。
- 尝试填满一个使用集群构建的分布式应用程序中的邮箱。尝试不间断地发送超大消息。当集群接收到过多消息时，观察集群的健康状况。当网络流量过大时集群的健康状况如何？
- 能否想出解决方案，解决观察到的网络饱和问题？
- 试着实现 Reactive Streams，看看能不能防止错误的发生（本书没有介绍 Reactive Streams，读者需要花些功夫来学习）。

7.5　小结

在日常工作中，我们可能不需要太多关注邮箱，因此把本章放在了本书的结尾部分。不过理解系统中邮箱的变化带来的影响是至关重要的。实现有消息数量限制的邮箱似乎是防止系统崩溃的好办法，不过我们希望能够尽快对用户做出响应(即使在失败情况下)，因此在系统达到性能极限时，相较于丢弃消息，快速返回失败通常是更好的选择。本章介绍了一些应用程序大规模运行并且达到性能极限时可以使用的工具。

在下章中，我们将学习行为驱动的测试与开发。此外，还将了解如何使用 Actor 和类进行领域建模。

第 8 章
测试与设计

虽然对测试和设计相关问题的介绍贯穿全书，不过都是在介绍其他具体工具的细节时顺带提及。既然现在已经介绍了 Akka 提供的诸多工具，现在就更详细地介绍设计和测试的一些通用方法。

测试和设计这两个话题看似毫不相关，但它们其实会互相影响。要编写容易测试的代码，就必须要有优秀的设计。同样，如果代码设计得很好，要对其行为进行测试自然也会比较简单。

在测试这个话题中，我们将介绍行为驱动开发（Behavior Driven Development，BDD）的基本概念，并学习如何使用这些方法提供 Actor 的文档并对 Actor 进行测试。优秀的测试方法是最优秀的提供文档并描述代码的工具之一。测试不会过时，也不会与应用程序的行为不符。一般来说，只要测试变得无效，测试结果就会失败。因此，这就强制工程师修改测试，使之与应用程序的行为相符。

在设计的话题中，我们将介绍 Actor 的代码层中的一些基本策略，这些策略可以简化对 Actor 行为的测试。除此之外，我们还会介绍如何在应用程序的不同部分之间创建上下文边界，使得应用程序在规模不断增长时能够保持灵活性以及可维护性。我们将学习何时应该创建这些边界以及如何创建这些边界，并了解边界与 Akka 扩展的关系。

在本章中，我们将介绍下述话题：

- 领域驱动设计以及如何使用 Actor 和类进行建模；
- 行为驱动的测试与开发。

8.1 示例问题

在本章中，我们将针对一个全新的问题，从头开始分析与设计，然后对其进行构建和测试。在本章的例子中，我们将了解如何构建一个聊天室应用程序的一部分。假设我们在一个机构中工作，接到了一个需求，要求我们编写一个在办公室内部使用的

软件。由于该软件只在私有网络内部使用，所以客户端可以通过 Akka Remoting 与服务器进行通信。

宏观地说，我们可能需要支持的功能如下所示：

- 应用要有一个大厅，显示不同的聊天室；
- 用户可以从大厅中加入或离开某个聊天室；
- 当用户加入聊天室时，可以接收到该聊天室的最近聊天历史；
- 显然，任何人在聊天室发布信息时，聊天室中的所有人都会收到更新。

用户客户端会有另一个团队来开发，然后与我们开发的应用程序进行交互。由于客户端是一个原生应用程序（Swing 应用程序），所以我们不需要提供 HTTP API，只要使用 Actor Remoting 和 Actor 进行通信即可。

8.2　应用程序设计

编写代码时可以选用多种已经被论证的方法。比较传统的是瀑布模型，首先进行设计与分析，然后编码。另一种方法是敏捷开发，特别是极限编程，这种方法最小化前期分析的过程，在开发过程中不断迭代，对代码进行重构，最终得到优秀的设计。

在现实中，两者的界限是比较模糊的，我们经常会先进行一些分析，然后在开发的过程中有了一些新发现，导致我们重新思考，并重新设计解决方案，直至最终完成项目。不过，据笔者观察，一些已经连续开发了好几年的项目，可以说开始一个新项目的时候，还是需要完成正确的程序基础设计，否则会在未来给开发带来很多的麻烦。

由于这里要开始一个全新的项目，因此应该在开始项目之前谨慎一些，先进行设计。我们将采用从上至下的方法，思考如何将应用程序切分为多个独立的模块，独立开发并测试每个模块。通过将问题域切分为独立的模块，可以确保单独解决的问题更简单。我们可以把每个模块分配给一个开发者来负责，而他/她可以在较短的时间内理解要解决的问题，无需从前到后理解整个系统。

现在就来设计聊天室应用程序。首先来看一下如何设计领域模型，以一种简单又清晰的方式来表示试图构建的应用。我们采用面向对象的方法，目的是把行为和数据封装到类中，并且使得互相交互的类之间不需要了解对方的内部细节。创建了有这些特点的类之后，就可以帮助我们将应用程序不断修改并扩展的过程中带来的花销降到最低。

通过观察简单的需求描述，很容易就能够识别出这个问题域中的一些主要实体：

- **聊天室**：聊天室包含一个可以作为唯一标识的主题（或者是名字）。
- **用户**：当用户注册时，会得到一个用户名。

- **大厅**：大厅包含聊天室列表。也可以包含在线用户列表。可能只有一个大厅，但是随着应用程序的扩展，我们也有可能把多个办公室通过网络连接起来，或是为不同的团队创建不同的大厅。

笔者还发现，从一开始就讨论一下要解决的问题中哪里可能会包含状态是很有帮助的。因为我们的环境是异步的，多个用户可以同时和服务器状态进行交互，所以对状态的思考有助于我们决定如何才能最有效地利用 Akka 来构建应用程序。下面就是可能会有状态的地方：

- **大厅**：包含所有聊天室的列表；
- **聊天室**：包含消息历史以及需要接收到聊天更新的当前用户列表。

我们知道，需要将状态封装在并发系统中时，Akka 是一个极佳选择。所以这些问题很可能可以用 Actor 来很好地实现。

架构设计

我们的聊天室将使用客户端/服务器模型。我们现在知道的是，用户会直接使用客户端，服务器端有大厅和聊天室。客户端、大厅和聊天室是独立的领域元素。围绕着三个元素建立清晰的上下文边界之后，我们就可以分别对它们单独分析，如图 8-1 所示。

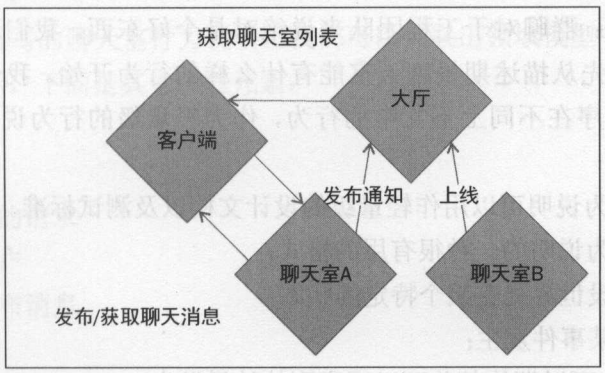


图 8-1

问题域中各组件之间的交互如下所示：

- 客户端从大厅获取聊天室列表；
- 当新聊天室上线时，发布通知表示该聊天室可用，令客户端能够查阅到新聊天室；
- 客户端可以加入聊天室；
- 客户端可以在聊天室中发布消息；
- 客户端可以接收到聊天室中消息更新。

我们稍后会将上面的行为转换为行为说明，不过通过列出基本的交互，我们可以识别出要解决的问题中包含的各组件。列出了各组件后，我们就可以划分出上下文边界，将各组件隔离开来。同样地，如果将各组件解耦，那么单独测试这些组件也变得很容易。

最后一个好处和 Akka 的位置透明性有关。如果我们将聊天室和大厅解耦，那么久可以在不同的服务器上运行聊天室。用户可以通过一个大厅获取所有可用的聊天室，而我们则可以根据需求对聊天室进行水平扩展。既然我们已经了解了各组件的整体架构设计，现在就可以选择一个组件进行集中设计了。

8.3　设计、构建并测试领域模型

设计、构建以及测试在开发过程中有着很大的重叠。有大量的方法可供我们采用。首先进行一些前期分析，描述领域模型及其行为，然后基于书面的分析抽取出领域模型，并编写测试用例。在下面的小节中，我们将专门讨论聊天室及其与用户的交互。

8.3.1　行为说明

假设读者曾经使用过某种类型的聊天室，比如 Slack、Hipchat、Google Hangouts 或是 Campfire。如果从未使用任何这类工具的话，应该评估一下是不是可以选择一个引入到开发团队中：群聊对于工程团队来说绝对是个好东西。我们现在只关注单独的聊天室设计，可以先从描述期望聊天室能有什么样的行为开始。我们可以用一些简单的语句描述应用程序在不同上下文中的行为，作为轻量级的行为说明，驱动后续的构建和测试。

基于行为的行为说明可以用作轻量级的设计文档以及测试标准。Given When Then 是可以用来编写行为说明的一种很有用的格式：

- Given：假设世界处在某个特定的状态；
- When：当某事件发生；
- Then：那么可以期待某些可以观察到的结果发生。

我们可以使用这种格式来描述应用程序的上下文或是状态、应用程序中发生的事件以及该事件发生后观察到的结果。读者可能会发现，编写单元测试时，也会非正式地分成这三步：初始化、执行操作以及断言。

对于一个聊天室 Actor 来说，我们可以定义一些简单场景下的行为，如下所示：

- **场景**：用户加入聊天室：
  - Given 聊天室中没有用户，When 聊天室收到一个用户要加入的请求，Then 聊天室应该将用户加入用户列表；

- Given 聊天室中有聊天历史，When 用户加入聊天室，Then 用户应该接收到聊天室中的最近 10 条历史消息；

- 场景：向聊天室中发布消息：
  - Given 聊天室中有已经加入的用户，When 聊天室接收到聊天室中的某个用户发来的要发布消息的请求，Then 聊天室应该更新聊天历史；
  - Given 聊天室中有已经加入的用户，When 聊天室接收到要发布消息的请求，Then 聊天室应该通知已经加入的用户。

这些行为说明只是最基本的，并不完整，不过已经足以说明其用处。我们稍后可以添加更多行为说明，并修改应用程序的行为，使之满足这些行为说明。

8.3.2 设计领域模型

既然我们已经有了书面的行为说明，实际上就可以在这个基础上进行设计和测试了。互相之间讨论问题域时使用统一的标准协议和语言的妙处在于它使得设计和命名变得相当简单。Eric Evans 在他的《领域驱动设计》（“Domain Driven Design”）一书中描述了一些我们和用户讨论应用程序时可以使用的术语，叫做领域通用语言（Ubiquitous Language）。领域通用语言有一个优点：它即表示了领域模型的结构，又给出了我们想要在模型中使用的名字。

重新阅读之前编写的聊天室行为说明，我们可以抽取出领域模型中的所有模块，用领域通用语言来表示，下面是其中一些元素：

- 聊天室
- 用户
- 加入聊天室的请求
- 已加入的用户
- 向聊天室发布消息
- 聊天历史

有一点令人难以置信：我们写下来的行为说明基本上已经描述了所有的领域模型。观察行为说明，我们可以抽取出用领域通用语言来标识的所有表述。剩下唯一要做的就是如何把它们组合起来。我们知道，在面向对象编程中，我们要尽可能地把行为和数据封装起来，基于这个目标，我们把相关的行为和数据封装在类中。我们还知道 Akka 可以用于处理状态，所以我们将基于这一点来理解使用 Actor 能够带来的好处。

我们可以做出如下推断：

- 聊天室有一个用户列表，可以通过 joinedUsers 引用访问，该列表可以修改（状态）；
- 聊天室有一个发布列表，叫做 chatHistory，该列表也可以修改（状态）；

- 加入聊天室的请求包含一个用户的引用。

由于聊天室本身包含状态，所以使用 Actor 可能会带来好处。其他任何东西都不需要包含任何可变状态，所以对于识别出的其他元素，都可以使用不可变对象。既然已经知道了领域模型大致的样子以及描述其行为的行为说明，我们现在就可以开始构建并测试了。

8.3.3　构建并测试领域模型

既然我们已经识别出了问题域中的元素，现在就可以开始构建并测试了。开发者经常会把要编写的部件及其单元测试作为任务单元一起完成，这会带来一些好处。如果在构建行为的时候就编写测试，就可以更容易地在测试中捕捉到所有可能发生的情况。所以当开发者完成所有任务时，就会发现已经编写了一系列顺利的单元测试，这些测试用例正好描述了应用程序的行为。如果读者从未使用过测试驱动开发以及行为驱动开发的话，不妨在工作中试一试这些方法。在很多时候，我们会先编写一些有用的测试用例，而不是在开发完成后再编写测试用例。

本书将要介绍的方法基于行为说明，将行为说明翻译成测试用例，然后再编写代码使得测试用例/行为说明可以通过。

首先，我们知道用户是远程的，所以可以先对远程用户进行建模。我们会在别的地方实现用户实体的定义，不过聊天室并不需要担心这一点。聊天室只需要知道将消息更新发送至哪里，以及要显示用户实体信息所需要提供的信息。既然如此，我们大可以现在就对用户实体进行建模。在这个例子中，我们将直接使用一个简单的 UserRef。

Java 代码：

```
public class UserRef {
    public final ActorRef actor;
    public final String username;
    public UserRef(ActorRef actor, String username) {
        this.actor = actor;
        this.username = username;
    }
}
```

Scala 代码：

```
case class UserRef(actor: ActorRef, name: String)
```

UserRef 包含要显示的名字以及一个 Actor 引用。我们可以将消息更新发送至该引用。这就是聊天室本身目前为止需要的所有用户信息。

聊天室还需要保存两个数据：已加入的用户以及聊天历史。由于这些数据表示状态，而且多个用户可能会同时试图访问并修改状态，所以我们认为使用 Actor 来实现聊天室可能会带来好处。因为聊天室会是一个 Actor，所以我们将发送消息的请求和加入聊天室的请求建模成不可变消息。我们将以一个动词开头来命名这两个消息（这可能和 Java 命名规范略有不同，但是消息名通常都这样命名）。这样可以很明确地表达出消息的作用。这样我们就可以在消息中包含任何与请求有关的数据了。

Java 代码：

```
public class Messages {
    static public class JoinChatroom {
        public final UserRef userRef;
        public JoinChatroom(UserRef userRef) {
            this.userRef = userRef;
        }
    }
    static public class PostToChatroom{
        public final String line, username;
        public PostToChatroom(String line, String username) {
            this.line = line;
            this.username = username;
        }
    }
}
```

Scala 代码：

```
case class JoinChatroom(userRef: UserRef)
case class PostToChatroom(line: String, username: String)
```

既然已经定义了消息，现在就可以编写聊天室的代码框架了。我们现在仅仅是创建 Actor，还不会实现任何逻辑。因为我们也已经知道了要包含哪些状态，所以现在也可以把这些成员变量添加到 Actor 中：

- 聊天室有一个叫做 joinedUsers 的用户引用列表，该列表可以改变（状态）；
- 聊天室有一个叫做 chatHistory 的消息列表，该列表也可以改变（状态）。

Java 代码：

```
public class Chatroom extends AbstractActor {
    List<Messages.PostToChatroom> chatHistory =
        new ArrayList<Messages.PostToChatroom>();
    List<UserRef>joinedUsers = new ArrayList<UserRef>();
    @Override
    public PartialFunction<Object, BoxedUnit> receive() {
```

```
        return ReceiveBuilder
            .matchAny(o -> System.out.println("received unknown message"))
            .build();
    }
}
```

Scala 代码：

```
class Chatroom extends Actor {
  var joinedUsers: Seq[UserRef] = Seq.empty
  var chatHistory: Seq[PostToChatroom] = Seq.empty

  override def receive = {
    case _ =>
      println("received unknown message ")
  }
}
```

8.3.4　基于行为说明编写代码

我们可以重新看一下编写的行为说明，在此基础上开始编写类以及测试用例。在这里挑选其中一个行为说明，展示编写测试用例的方法。接着，将实现具体的行为。在实现类的时候，每次添加与一个行为说明对应的逻辑。在完成了与一个行为说明对应的代码之后，我们将介绍测试 Actor 的不同方法。

场景：用户加入聊天室：

- Given 聊天室中没有用户，When 聊天室收到一个用户要加入的请求，Then 聊天室应该将用户加入用户列表

在 Java 中，我们将继续使用 JUnit。不过如果开始编写新项目的话，可以看一下其他通过适当的注解来支持 Given、When、Then 风格的行为说明测试框架。

```
public class ChatroomTest extends TestCase {
    static ActorSystem system = ActorSystem.apply();

    @Test
    public void testShouldAddUserToJoinedUsersWhenJoiningTest() {
        //Given a Chatroom has no users
        Props props = Props.create(Chatroom.class);
        TestActorRef<Chatroom> ref =
            TestActorRef.create(system, props, "testA");
        Chatroom chatroom = ref.underlyingActor();
        assertEquals(chatroom.joinedUsers.size(), 0);
        //When it recieves a request from a user to join the chatroom
```

```
        UserRefuserRef = new UserRef(system.deadLetters(), "user");
        Messages.JoinChatroom request = new Messages.JoinChatroom(userRef);
        ref.tell(request, system.deadLetters());

        //It should add the UserRef to its list of joined users
        assertEquals(chatroom.joinedUsers.get(0), userRef);
    }
}
```

值得注意的是，Dan North 曾经提到，我们可以用单词“should”作为测试方法的前缀，这有助于提高测试方法名字的表达能力，可以在测试运行时打印出有用的信息。就像在注释中标注的那样，该测试的实际结构与我们的 Given When Then 格式对应。注释本身并不是必须的，但是我们在这里加上了注释以说明代码和行为说明之间的关系。

在 Scala 中，通用测试工具提供了更好的 DSL，所以测试的结构可以直接与行为说明对应起来：

```
class ChatroomSpec extends FunSpec with Matchers {

  val system = ActorSystem()
  describe("Given a Chatroom has no users") {
    val props: Props = Props.create(classOf[Chatroom])
    val ref: TestActorRef[Chatroom] =
      TestActorRef.create(system, props, "testA")
    val chatroom: Chatroom = ref.underlyingActor
    chatroom.joinedUsers.size should equal(0)

    describe("when it receives a request from a user to join the chatroom") {
      val userRef: UserRef = new UserRef(system.deadLetters, "user")
      val request: JoinChatroom = JoinChatroom(userRef)
      ref ! userRef
      it("should add the UserRef to the list of joined users") {
        chatroom.joinedUsers.head should equal(userRef)
      }
    }
  }
}
```

上面的代码展示了如何使用 FunSpec 来组织测试用例的结构，可以直接与 Given When Then 的格式一一对应。虽然上述代码很直观地反映了行为说明，但是在真实情况下，笔者会重复一些代码，避免创建太多嵌套结构，增加测试的可读性。和生产环境中的代码不同，在测试中是可以包含一些重复代码的。使用 ScalaTest 时，有多种方法可以用来组织测试代码的结构。请查阅 API，选择最适合的方法。

现在，需要运行测试用例，确认其失败，并输出我们期待的问题。运行失败的测试用例是很重要的一个步骤，它使得我们能够确认测试用例确实在验证需要验证的代码。在这里可以看到测试用例失败的原因是列表中没有已经加入的用户，和我们期待的一致。接着，当测试用例通过的时候，我们就知道已经根据行为说明正确地实现了相关的行为：

```
[info]   - should add the UserRef to the list of joined users ***
FAILED ***
[info]     java.util.NoSuchElementException: head of empty list
```

现在我们已经介绍了设计并测试一个新项目的基本知识。回顾一下，可以通过对问题的描述方式推导出很多设计要点。我们识别出可能存在状态的地方，而如果识别出的状态在并发环境中，就可以决定使用 Actor 来实现。而如果某个行为不包含会被多个地方同时访问的可变状态，那么我们也许就可以直接使用普通的对象来实现。例如，也许可以用一个类来表示 ChatHistory，而不是直接使用列表。由于 ChatHistory 只会被聊天室访问，而且聊天室 Actor 每次只会处理一个消息，所以 ChatHistory 可以是一个普通的对象，实现一个简单的非同步方法 getRecentHistory。

在确认测试用例失败之后，我们就可以实现行为了，接着使用传统的 TDD/BDD 方法确认测试运行通过。然后继续使用相同的方法实现下一个行为说明：先创建测试用例，然后实现行为。发现并编写的每一种情况都应该有对应的测试用例。

笔者认为，这是编写易于维护的代码最可靠的方法。

接下来，我们将学习 akka-testkit 模块中的一些不同功能，这些功能可以帮助我们编写出清晰并且包含很多有效信息的测试用例。

8.4　测试 Actor

Akka-testkit 模块是 Akka 的一部分，它提供了一些可以用于测试 Actor 的工具。我们将扩展聊天室的例子，学习如何使用 testkit 中的工具优化我们的测试。

我们将介绍两个主要的内容：对 Actor 行为的单元测试以及对 Actor 响应的测试。

8.4.1　测试 Actor 行为及状态

前文已经介绍过同步的测试用例，在这里回顾一下，我们可以使用 TestActorRef 对 Actor 的请求和响应进行测试，并且无需使用 await。使用 await 是一种更灵活、更现实的测试 Actor 的方法，不过如果要单独测试 Actor 的行为，那么还有好多方法可以使用。

在前面的例子中，注意到我们并没有等待消息处理完成。因为我们使用 Akka Testkit 的 TestActorRef 来创建 Actor，Actor 会使用发起调用的线程（通过 Calling ThreadDispatcher）来处理消息，所以我们自然而然地可以马上通过 TestActorRef 创建的 Actor 来进行同步测试。这是 Akka Testkit 提供的第一个便利之处，我们在前面已经做了介绍。

除此之外，akka-testkit 还支持访问 ActorRef 指向的对象，这就允许我们使用通常无法使用也不想要使用的方式来访问该对象。如果没有 akka-testkit，我们就需要发送消息，检查 Actor 的内部状态，验证测试结果。这样可能就必须要为了能够正确地进行 Actor 的测试而添加一些功能，这可不是我们想要做的。如果可以访问到实际的 Actor，那么就可以用最少的代码来测试该 Actor 的状态，也不用做多余的专门用于测试的设计。

现在我们就来看一下如何使用 testkit 的这一功能改写使用 TestActorRef 的 Actor 代码的测试用例。Akka 的 TestActorRef 给我提供了一种访问实际 Actor 的方式，而且如果我们可以访问到实际的 Actor，就可以直接验证该 Actor 向外暴露的任意成员变量和方法。但是如果创建 Actor 时把一些行为实现在方法里，而不是在 receive 块中，那么实际上可以像测试普通类一样测试 Actor。如果想要创建清晰而又简单的测试用例来测试 Actor 的行为，而且不想发送和接收消息，那么这种方法确实带来了很大的好处。我们也想验证集成测试的结果，不过如果仅仅是要测试行为的话，我们可以直接测试实际的对象。回到之前创建的测试用例，我们来看一下如何访问实际的 Actor 来给我们的测试带来便利。

首先，我们来看一下上一个行为说明的实现：

- **场景**：用户加入聊天室：
  - Given 聊天室中没有用户，When 聊天室收到一个用户要加入的请求，Then 聊天室应该将用户加入用户列表。

我们在 Actor 中编写一个方法来表示该行为。

Java 代码：

```
public class Chatroom extends AbstractActor {
    List<Messages.PostToChatroom>chatHistory =
        new ArrayList<Messages.PostToChatroom>();
    List<UserRef>joinedUsers = new ArrayList<UserRef>();
    @Override
    public PartialFunction<Object, BoxedUnit> receive() {
        return ReceiveBuilder
            .match(Messages.JoinChatroom.class, this::joinChatroom)
            .matchAny(o -> System.out.println("received unknown message"))
            .build();
    }
```

```
    public void joinChatroom(Messages.JoinChatroommsg) {
        joinedUsers.add(msg.userRef);
    }
}
```

Scala 代码：

```
class Chatroom extends Actor {
  var joinedUsers: Seq[UserRef] = Seq.empty
  var chatHistory: Seq[PostToChatroom] = Seq.empty
  override def receive: Receive = {
    case x: JoinChatroom =>
      joinChatroom(x)
    case _ =>
      println("unimplemented")
  }
  def joinChatroom(joinChatroom: JoinChatroom) {
    joinedUsers = joinedUsers :+ joinChatroom.userRef
  }
}
```

注意到我们在一个方法内实现了行为逻辑，而没有直接在 receive 块中实现。这样一来，就无需担心消息的真正传输，直接测试并修改行为了。接着，我们可以在测试中使用 akka-testkit 的 TestActorRef 来获取实际的 Actor 并测试其行为。

Java 代码：

```
@Test
public void testShouldAddUserToJoinedUsersWhenJoiningTest() {
    Props props = Props.create(Chatroom.class);
    TestActorRef<Chatroom> ref = TestActorRef.create(
        ActorSystem.apply(), props, "testA");
    Chatroom chatroom = ref.underlyingActor();
    UserRefuserRef = new UserRef(system.deadLetters(), "user");
    Messages.JoinChatroom request = new Messages.JoinChatroom(userRef);
    chatroom.joinChatroom(request);
    assertEquals(chatroom.joinedUsers.get(0), userRef);
}
```

Scala 代码：

```
describe("Given a Chatroom has no users (Unit example)") {
  val props: Props = Props.create(classOf[Chatroom])
  val ref: TestActorRef[Chatroom] =
    TestActorRef.create(ActorSystem(), props, "testA")
```

```
    val chatroom: Chatroom = ref.underlyingActor
    chatroom.joinedUsers.size should equal(0)

    describe("when it receives a request from a user to join the chatroom") {
      val userRef: UserRef = new UserRef(system.deadLetters, "user")
      chatroom.joinChatroom(JoinChatroom(userRef))
      it("should add the UserRef to the list of joined users") {
        chatroom.joinedUsers.head should equal(userRef)
      }
    }
  }
```

从这个例子中我们可以看到，虽然代码并没有简单很多，但是我们不再发送消息，而是直接调用方法。如果想要测试 Actor 内的多个方法，那么这种做法可以大大地帮助我们解耦，更细粒度地测试 Actor 行为的细节。

8.4.2 测试消息流

对 Actor 中行为的测试固然是有用的，但是如果想要在更复杂的情况下测试消息的发送与接收，该怎么做呢？如果想要模拟另一个 Actor 的行为，确认 Actor 之间能够顺利集成，又该如何呢？

接下来我们将介绍一些涉及到其他 Actor 的场景，以及两个主要的方法：将测试类作为 Actor 以及用一个 TestProbe 来模拟 Actor

将测试作为 Actor

现在，我们将介绍如何令测试类接收消息，这样我们就不需要在使用额外的外部 Actor 来检查被测试 Actor 的消息流是否正确。首先，我们将使用下面的行为说明。

- 场景：用户加入聊天室：
  - Given 聊天室中有聊天历史，When 用户加入聊天室，Then 用户应该接收到聊天室中的最近 10 条历史消息。

在这种情况下，我们想要确认当用户加入聊天室时，能够从消息历史中接收到最近 10 条更新。这一过程涉及到两个 Actor：聊天室 Actor 和 UserRef 引用指向的用户 Actor。由于我们并不关注用户 Actor 的任何实现细节，只需要测试聊天室 Actor 发送的消息数据是否正确。对两个 Actor 之间的交互进行测试的最简单的方法就是将测试本身作为接收方。Akka-testkit 支持测试用例本身接收消息，所以想要测试响应时，可以使用 akka-testkit 提供的一个 API 来表示断言：

```
@Test
public void testShouldSendHistoryWhenUserJoin() {
    new JavaTestKit(system) {{
        //Given
        Props props = Props.create(Chatroom.class);
        TestActorRef<Chatroom> ref = TestActorRef.create(system, props);
        Chatroom chatroom = ref.underlyingActor();
        Messages.PostToChatroom msg =
            new Messages.PostToChatroom("test", "user");
        chatroom.chatHistory.add(msg);
        //When
        UserRef userRef = new UserRef(system.deadLetters(), "user");
        Messages.JoinChatroom request = new Messages.JoinChatroom(userRef);
        ref.tell(request, getRef());

        //Then
        List expected = new ArrayList<Messages.PostToChatroom>();
        expected.add(msg);
        expectMsgEquals(duration("1 second"), expected);
    }};
}
```

在 Java 代码中，要注意的一点是，我们发送消息时将测试本身的 TestActorRef 作为发送方：ref.tell(request, getRef())，然后测试本身又使用某个 expectMsg*方法来测试其收到的消息。我们在这里检查两者是否相等，不过读者也可以测试类的类型是否相等。

在 Scala 中，我们没有像在 Java 中一样创建一个匿名的 TestKit 对象，直接修改测试类的定义即可，语法更优雅。

我们必须要修改测试类的定义：

```
classChatroomSpec(_system: ActorSystem) extends TestKit(_system)
  with ImplicitSender
  with Matchers
  with FunSpecLike {
```

这样测试类本身就变成了一个 Actor，所以行为说明的实现如下所示：

```
describe("Given a Chatroom has a history") {
  val props = Props.create(classOf[Chatroom])
  val ref = TestActorRef.create(system, props)
  val chatroom: Chatroom = ref.underlyingActor
  val msg = PostToChatroom("test", "user")
  chatroom.chatHistory = chatroom.chatHistory.+:(msg)

  describe("When a user joins the chatroom") {
    val userRef = UserRef(system.deadLetters, "user")
```

```
    val request = JoinChatroom(userRef)
    ref ! request

    it("(the user) should receive the history") {
      expectMsg(1 second, List(msg))
    }
  }
}
```

这要比 Java 的语法更清晰一点：我们知道测试类本身是一个 Actor，然后调用 tell 的时候，测试类本身会隐式作为发送方，因此可以接收到响应。唯一要提一下的一点是 expectMsg 方法：该方法会暂停测试，等待相应的响应。如果在传入方法的时间限制内没有收到期待的消息，那么测试就会失败。

将 TestProbe 作为模拟 Actor

接下来，我们将使用向聊天室发布消息的行为说明，看看如何在测试中模拟一个外部的 Actor。

- **场景**：向聊天室中发布消息：
  - Given 聊天室中有已经加入的用户，When 聊天室接收到要发布消息的请求，Then 聊天室应该通知已经加入的用户。

我们可以把已经加入的用户作为一个模拟 Actor，该模拟 Actor 期待接收一条消息。下面是代码。

Java 代码：

```
@Test
public void testShouldSendUpdateWhenUserPosts() {
    //Given
    Props props = Props.create(Chatroom.class);
    TestActorRef<Chatroom> ref = TestActorRef.create(system, props);
    Chatroom chatroom = ref.underlyingActor();
    final TestProbe probe = new TestProbe(system);
    UserRef userRef = new UserRef(probe.ref(), "user");
    chatroom.joinChatroom(new Messages.JoinChatroom(userRef));

    //When
    Messages.PostToChatroommsg =
        new Messages.PostToChatroom("test", "user");
    ref.tell(msg, probe.ref());
    //Then
    probe.expectMsg(msg);
}
```

Scala 代码：

```
describe("Given a Chatroom has a joined user") {
  val props = Props.create(classOf[Chatroom])
  val ref = TestActorRef.create(system, props)
  val chatroom: Chatroom = ref.underlyingActor
  val probe: TestProbe = new TestProbe(system)
  val userRef: UserRef = new UserRef(probe.ref, "user")
  chatroom.joinChatroom(JoinChatroom(userRef))

  describe("when someone posts to the chatroom") {
    val msg = PostToChatroom("test", "user")
    ref.tell(msg, probe.ref)
    it("(joined user) should get an update") {
      probe.expectMsg(msg)
    }
  }
}
```

可以发现，上面的代码和之前的测试代码很类似，不同点在于发送消息时将 TestProbe 传入作为发送方，接着进行断言的时候也是基于 TestProbe 的（probe.expectMsg）。TestProbe 提供了一些额外的功能，比如发送消息，所以绝对可以使用 TestProbe 来模拟任意 Actor 行为并且对接收到的响应进行断言。在使用 Actor 时， TestProbe 是一个非常强大的工具。

8.5　测试建议

上文已经从不同的层次介绍了一些测试技术。首先，介绍了如何对 Actor 中的代码进行单元测试，然后从测试用例与 Actor 之间发送消息的角度介绍了如何对 Actor 进行测试，最后介绍了如何创建 TestProbe，模拟一个 Actor 与被测试的 Actor 进行交互。而且我们当然可以使用 TestProbe 来测试许多不同的集成场景。

然而要明确对哪些代码需要进行测试，对哪些代码无需进行测试却没这么简单。

笔者见过的最好的建议可能就是 Kent Beck 在 Stack Overflow 上关于测试覆盖率的一个回复。他说：“别人付给我钱是因为我编写了解决问题的代码，而不是因为我编写了测试用例，所以我的哲学就是，在能够达到特定置信度的前提下尽可能少地编写测试用例。”可以从 http://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests 查看这个问题。

从这个问题的回答中，可以总结出于 Actor 测试有关的一点：测试最重要的部分。

我们没有必要为了确认Actor运行无误并避免引入回归测试而同时测试Actor的行为以及集成效果。只要有针对性的测试即可。测试那些必须正确的地方。

由于TestActorRef使用了CallingThreadDispatcher，所以它提供了一种同步测试Actor的机制，无需在使用TestProbe或是直接将测试用例作为发送方时调用在expectMsg之后调用Thread.sleep。使用这样的机制要比直接在测试用例中调用Thread.sleep更好，这样可以保持测试用例快速运行。当我们有一整个测试集要跑的时候，尽可能减少运行时间很重要的。如果测试集中的好几个测试用例都调用了Thread.sleep，那么运行流的中断或暂停运行会使得整个测试集的运行时间变得更长。

要编写运行速度较快的测试用例可能没那么简单，不过我还是推荐大家在编写测试用例时多考虑运行速度的问题，毕竟我们希望整个团队能够依赖这些测试用例来确认代码的质量。要了解如何在不调用Thread.sleep的前提下运行测试用例——使用Akka提供的工具基本上都能完成这一点。

如果使用了类似Thread.sleep的机制，那么需要消息难度另外一点就是测试的不稳定性。编写出的测试用例很可能平常都能通过，但是有时候就会突然失败。如果有人接手了我们的代码，却随机地发现测试失败，那么他们很可能会试图分析是不是代码中出了什么问题。而当测试线程需要等待外部事件时，就可能会导致间断性的测试失败。所以最好从新编写测试用例，把测试或是TestProbe就嵌在行为逻辑之中，这样就能在事件发生时捕捉事件，进行断言，而不用等待任意的一段时间再验证结果。在比较复杂的情况下，可以使用类似Java中的CountDownLatch来等待事件的完成。

如果没有什么编写测试的经验，那么我只想在这里指出一点：生产环境中代码的目标在编写测试时可以放宽一些。具体来说，测试中可以包含重复的代码。如果为每个行为说明都需要编写一个类似的初始化操作可以使得API行为的文档更为清晰，那么别担心，并没有必要删除这些重复代码。

8.6 课后作业

为了确保理解如何使用Akka提供的工具，笔者推荐读者实现本章中介绍过的以及下面的行为说明：

- 确认当用户发布消息时，可以收到一个OK作为响应。
- 确认当用户加入聊天室并发布消息时，该用户自己不会收到消息更新
- 继续设计应用程序。如果用一个类实现ChatHistory而不是一个简单的列表，那么能否只用一个不同的类呢，使用Actor能带来什么好处呢，又会带来什么坏处呢？

8.7　小结

使用 Actor 进行设计以及对 Actor 的测试的学习曲线稍微有一点陡峭。关于 Actor，要意识到的最重要的一点就是使用 Actor 并不总是最好的方案——如果使用普通类更简单的话，就应该使用普通类。不过在状态和并发同时存在的问题域中，Actor 通常是比较好的解决方案。相较于普通的类，Actor 为应用程序的构建以及测试提供了更多的功能。Akka 提供了许多优秀的工具，帮助我们构建并测试 Actor，而本章作为全书的结尾部分，介绍了其中的一些工具。这些工具不仅仅有助于设计和测试的过程，还可以更好地表达测试用例的含义。

在下一章中，我们将介绍 Akka 中的其余一些特性，其中有一些和部署以及监控有关。我们还会介绍学习完本书之后，读者下一步可以继续学习哪些内容。

第9章 尾声

我们已经到了旅途的终点。现在我们已经了解了不少使用 Scala 或 Java 8 以及 Akka 来编写并发分布式系统的知识了。

本书的读者可能背景不尽相同——有的读者有一些分布式计算的经验，有的则没有相关经验。不论是哪一种，要掌握本书介绍的知识都需要进行实践。

Akka 工具集中提供的工具很多，文档也非常详尽，可以查阅文档了解 Akka 提供的所有功能。本书的目的是告诉读者什么时候使用 Akka 的不同工具，以及为什么要使用这些工具。本章将介绍一些读者可能会有兴趣的重要功能和模块，以及下一步可以学习的知识，如下所示：

- 其他 Akka 功能及模块：
  - 日志
  - EventBus
  - Agent
  - Akka 持久化
  - Akka I/O
  - Akka Streams 与 Akka HTTP
- 下一步：
  - 学习模型驱动设计
  - 部署工具
  - 监控日志与事件

9.1 其他 Akka 功能及模块

本书一开始就说过，书中不会涉及 Akka 工具集的所有内容，而是会把重点放在帮助读者学习如何使用 Akka 处理分布式计算的问题。Akka 是一个很大的工具集，我们已

经介绍了所有重要的核心部分。

既然已经接近了尾声，现在我们就可以指出 Akka 工具集中读者可能会想更深入学习的内容。本书在这里对这些知识点进行简要介绍，帮助读者知道存在这些有趣的扩展功能。

9.1.1　Akka 中的日志

本书简要地介绍过 Akka 中的日志，不过我们在这里会重新回顾并展示如何使用更高级的功能。Akka 默认会将日志输出到控制台，不过它也会 slf4j 提供了一个事件句柄，我们可以将一个 slf4j 后台引入项目中（如 Logback）来使用该句柄。

要使用 slf4j 的 logger，就需要提供 Akka 的 slf4j 模块（并不包含在 Akka core 当中），以及一个类似 Logback 的 slf4j 后台。可以使用下述代码将这两个依赖加入 build.sbt 文件中：

```
"ch.qos.logback" % "logback-classic" % "1.0.0" % "runtime",
"com.typesafe.akka" %% "akka-slf4j" % "2.3.11"
```

接着，可以在应用程序的配置文件（application.conf 文件）中声明 Akka slf4j 事件句柄的使用方法，如下所示：

```
akka {
    event-handlers = ["akka.event.slf4j.Slf4jEventHandler"]
    loglevel = "DEBUG"
}
```

可以把一个 logback.xml 文件放到项目的 resources 文件夹下，通过 appender 来提供细粒度的功能更强大的日志控制。如果既想把日志保存到文件，又想把日志输出到控制台，那么基本的配置如下所示：

```
<configuration>
<appender name="FILE" class="ch.qos.logback.core.FileAppender">
<file>logs/app.log</file>

<encoder>
<pattern>%date %level [%thread] %logger{10} [%file:%line] %msg%n</
pattern>
</encoder>
</appender>
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
<encoder>
<pattern>%msg%n</pattern>
```

```
</encoder>
</appender>

<root level="debug">
<appender-ref ref="FILE" />
<appender-ref ref="STDOUT" />
</root>
</configuration>
```

要在应用程序中使用 Akka 的日志功能，可以显式地创建一个 logger，创建时传入一个 ActorSystem 对象：

```
//java
LoggingAdapter log = Logging.getLogger(getContext().system(), this);

//scala
val log = Logging(context.system, this)
```

如果使用 Scala 的话，可以将 ActorLogging Trait 混合进 Actor：

```
class MyActor extends Actor with akka.actor.ActorLogging {
  log.info("actor startup: {}", self.toString)
}
```

注意到 Akka 会接收一个变长参数列表作为消息参数，这些参数会按照顺序替代引号中的大括号{}。把异常当做第一个参数传入时，就可以将异常输出至日志：

```
#+BEGIN_SRC
//java
log.debug("params {} {} {}", param1, param2, param3);
log.error(e, "exception encountered: "); //exceptions are first arg

//scala
log.debug("params {} {} {}", param1, param2, param3)
log.info(e, "exception encountered: ") //exceptions are first arg
#+END_SRC
```

由 logger 来处理字符串插值可以提供性能上的好处。它只会在 logger 根据日志级别的设置真正会将事件输出日志时才会执行字符串插值。如果事件不会输出至日志，那么就不会执行字符串插值。因此，这种做法一般来说要比下面的做法更好：

```
#+BEGIN_SRC
log.debug("this actor is " + self().toString);
#+END_SRC
```

无论消息是否会被输出到日志，上面的代码都会在内存中创建一个连接后的字符串。

上面就基本上就覆盖了关于 Akka 中的日志我们需要知道的所有内容了。如果想了解日志配置的更多信息，可以查阅 Logback 的文档。通过配置 Logback（具体来说就是使用 logback.xml），可以打包日志文件、支持多个 appender 以及其他高级特性。Logback 还支持使用 groovy 语言来编写配置文件，用于一些特别高级的使用场景。

9.1.2　消息信道与 EventBus

Akka 中的 EventBus 对象可以用于发布和订阅事件，支持向多个 Actor 发送消息。我们可以用很少的代码来实现发布/订阅机制：向 Actor 发送订阅消息，接收订阅消息的 Actor 保存一个 Actor 列表。不过 Akka 也提供了一个处理这种情况的机制，可以监听一个话题。

EventBus 是 Akka 核心库的一部分，所以不需要再引入别的依赖。

要使用 EventBus，需要选择并扩展一个分类器，该分类器描述了事件的类型以及区分订阅者的方式。本小节末尾引用的文档对分类器做了详细描述。

要记住，我们始终可以使用 actorSelection 通过 Actor 的层级结构来发送消息。而如果要处理特定的话题订阅，不适合使用 Actor 层级结构，那么可以考虑 EventBus。

使用 EventBus 时，主要有三种类型需要考虑，我们将把它们声明称 Scala 的类型，或是定义成 Java 中的类型参数，如下所示：

- **分类器类型**：消息的话题类型
- **事件类型**：发布的实践中使用的数据类型：
  - 需要一个话题（由我们定义话题的逻辑）；
  - （可选）需要一些其他数据来发布事件。
- **订阅者类型**

可以使用多种分类器。下面是一个简单的例子，假设我们有一个叫做 EventBusMessage 的消息，我们希望使用 EventBus 令 Actor 订阅并发布该消息。EventBus 并不需要使用 Actor——它可以实现我们需要的任何发布/订阅行为。一个简单的 LookupClassifier 对象定义了事件的类型、话题/分类器类型以及为订阅者查询话题分类器的方法。在 LookupClassifier 中，订阅者需要是有序的。

一个简单的 Java 消息如下所示：

```
public class EventBusMessage {
    public final String topic;
    public final String msg;
```

```
    public EventBusMessage(String topic, String msg) {
        this.topic = topic;
        this.msg = msg;
    }
}
```

下面是 Java 和 Scala 的 EventBus 例子：

```
public class JavaLookupClassifier extends
  LookupEventBus<EventBusMessage, ActorRef, String> {

  @Override public String classify(EventBusMessage event) {
      return event.topic;
  }

  @Override public void publish(EventBusMessage event,
      ActorRef subscriber) {
      subscriber.tell(event.msg, ActorRef.noSender());
  }

  @Override public int compareSubscribers(ActorRef a, ActorRef b) {
      return a.compareTo(b);
  }

  // determines the initial size of the index data structure
  @Override public intmapSize() {
      return 128; }
  }
}

class ScalaLookupClassifier extends EventBus with LookupClassification {
  type Event = EventBusMessage
  type Classifier = String
  type Subscriber = ActorRef

  override protected def classify(event: Event): Classifier =
    event.topic

  override protected def publish(
    event: Event, subscriber: Subscriber): Unit = {
    subscriber ! event.msg
```

```
  }

  override protected def compareSubscribers(
    a: Subscriber, b: Subscriber): Int =
    a.compareTo(b)

  //initial size of the index data structure
  override protected defmapSize: Int = 128
}
```

上面的代码实现了一些相当基本的逻辑，比如如何确定话题，如何向任意相关订阅者发布事件并实现了如何比较订阅者需求。我们还需要声明 Map 的初始大小，不过它会根据需要自动增长。

使用方法如下所示：

```
JavaLookupClassifier lookupBus = new JavaLookupClassifier;
lookupBus.subscribe(myActor, "greetings");
lookupBus.publish(new EventBusMessage("time", System.
currentTimeMillis().toString));
lookupBus.publish(new EventBusMessage("greetings", "hello"));

val lookupBus = new JavaLookupClassifier
lookupBus.subscribe(myActor, "greetings")
lookupBus.publish(new EventBusMessage("time", System.
currentTimeMillis().toString))
lookupBus.publish(new EventBusMessage("greetings", "hello"))
```

在这个例子中，Actor 订阅了“greetings”这个话题。接着我们向话题“time”以及话题“greetings”分别发布一个事件。不出所料，Actor 只会接收到发布到话题“greetings”的消息。

Akka 文档提供了分类器的例子和解释，读者可以查阅文档深入研究如何使用 Akka EventBus 的特性。

9.1.3　Agent

Akka 的 Agent 模块灵感来自于 Clojure 的 Agent。Clojure 的 Agent 是响应式编程的基本组件，用于处理对于共享状态的访问。一个更易于理解 Agent 的类比是 Java 的 AtomicInteger，只不过可以用于任意值或类型。Agent 包含状态，并且允许我们对其存储的值以线程安全的方式执行原子操作，允许安全地读取该值。

更正式地说，Agent 为单个值提供了一个存储位置，并且提供了一个函数允许对该值的修改。Agent 允许对存储值进行安全的原子事务访问，这就提供了一种和 Actor 类似的支持安全并发访问状态的机制。

Agent 的优势是比使用 Actor 来封装状态更轻量级一些。如果需要安全处理单个包含状态的值或对象，那么可以考虑使用 Agent 是否合适，尤其是在考虑采用原子操作的时候（也就是说，首先取值，然后在满足特定的条件时赋值，而这一过程中出现的竞争条件可能会导致问题）。因此，当我们需要从多个线程访问单个值的时候，Agent 提供了除了 Actor 之外的另一种好方法。

Agent 并不是 Akka 的核心功能，它在另一个单独的模块中，所以我们需要将该模块的依赖添加到项目的 build.sbt 文件中：

```
"com.typesafe.akka" %% "akka-agent" % "2.3.6"
```

银行账号提款是一个相当标准的例子。假设我们首先需要查看银行账户中是否有足够的钱，如果足够的话，就可以提款。因为可能会有两个线程同时查看余额并从该账户提款，所以先查看余额再提款的操作需要是一个原子操作，这样就能确保某个线程查看余额后余额不被另一个线程修改。

如果多个线程能够同时访问一个共享的整数，那么就可能发生这样的事情：两个人同时试图从账号中取出 20 美元：

1. 账户中还有 25 美元。
2. 丈夫和妻子同时试图取出 20 美元。
3. 丈夫查看余额是否多于 20 美元。账户中还剩 25 美元。
4. 妻子查看余额是否多于 20 美元。账户中还剩 25 美元。
5. 丈夫提取 20 美元，账户余额设置为 5 美元。
6. 妻子提取 20 美元，将账户余额设置为 5 美元（操作非法！）。

由于两个线程在查询余额的时候账户中余额充足，所以两个线程都能够继续进行提款。但是当第二个线程提款时，它并不知道另一个线程已经从账户中提走了钱。银行现在少了 20 美元，并且不知道这钱去哪了！两个应用程序看上去都一切正常，但是如果两个事务同时发生，钱就会不翼而飞，而且没人发现。如果在有人发现这个问题之前这一现象发生了成百上千次，银行有可能已经损失了好多美元了！

我们需要把查询余额以及提款操作设置为原子操作（一个完整的工作单元），提供安全的功能。由于 Actor 每次只会处理一个消息，所以我们当然可以使用 Actor 来处理这个情况。不过使用 Agent 来解决这个问题要更简明一些。

Agent 的底层实现使用了 Actor 和线程，所以我们需要给 Agent 提供执行上下文。我

们将创建一个 Actor 系统，并使用该 Actor 系统的 dispatcher 来创建一个 Agent，该 Agent 中包含一个含有 25 美元的账户：

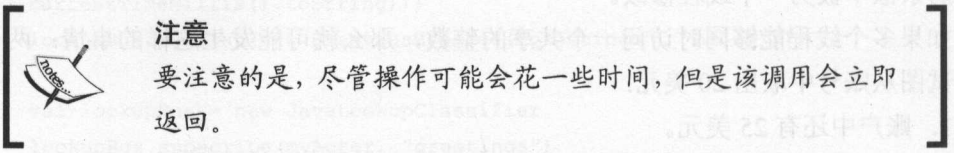
```
import akka.actor.ActorSystem;
import akka.agent.Agent;

//Java
ActorSystem system = ActorSystem.create();
Agent<Integer> account = Agent.create(25, dispatcher);

//Scala
val system = ActorSystem()
implicit valec = system.dispatcher
val account = Agent(25)
```

可以使用 Agent.get()或是 Agent.apply 来获取值：

```
Integer currentValue = account.get();
valcurrentValue = account()
```



注意

要注意的是，尽管操作可能会花一些时间，但是该调用会立即返回。

要通过该原子操作更新值，可以给 send 函数传入一个描述操作的函数（int => int）。在 Java 中，需要提供一个 Mapper 函数：

```
//Java
final Integer ammountToWithdraw = 20;
account.send(new akka.dispatch.Mapper<Integer, Integer>() {
    public Integer apply(Integer i) {
        if(ammountToWithdraw <= i)
            return i - ammountToWithdraw;
        else
            return i;
    }
});

//Scala
val ammountToWithdraw = 20
```

```
account.send { i =>
  if(i >= 20) {
    i - 20
  } else I
}
```

这些操作都是 fire-and-forget 风格的，它们运行在另一个线程池中，所以如果想要从 Agent 取回操作的结果，需要完成 Future。Alter 方法也是一样的，只不过 Future 中返回的是操作的结果。

使用 Scala 来完成这一操作提供了更多的灵活性。在 Scala 中，可以使用一个事务块，允许多个 Agent 参与同一个原子操作，然后返回该块的结果。假设我们想要从某个账户向另一个账户转账，那么可以使用事务块来实现多个 Agent 之间的交互，如下所示：

```
import scala.concurrent.stm._
val wifeAccount = Agent(25)
val husbandAccount = Agent(0)
val wasSuccess = atomic { txn =>
  if(wifeAccount() >= 20) {
    wifeAccount.send(_ - 20)
    husbandAccount.send(_ + 20)
    true
  } else false
}
```

Agent 是我们可以选用的一个比较有用的简单抽象。可能会使用 AtomicInteger 的地方，现在我们有了一个 AtomicAnything 对象，叫做 Agent。

9.1.4 Akka Persistence

读者可能感兴趣的另一个模块就是 Akka Persistence。Akka Persistence 提供了一种机制，支持在程序崩溃、JVM 重启或是 Actor 被监督者重启时保存 Actor 状态。

这个库的名字很容易令人误解，让人觉得它提供了一种类似数据库和键值存储的外部持久化机制。我们不会使用 Akka Persistence 来存储例如用户信息或账户信息这样的信息。

要记住，默认情况下，当 Actor 重启时，所有内部状态都会丢失，只会保留构造函数参数。Akka Persistence 能够保存 Actor 遇到过的事件，这样在 Actor 重启时，就可以重放这些事件，恢复 Actor 的状态。通过重放这些事件，Actor 可以快进，重新收集任意内部状态。例如，如果我们在编写一个用于性能测试的库，那么有了 Akka Persistence 以后，保存在内存中的计数器就能够更精确地收集某个端口的延时信息。如果 Actor 重启的话，就可以重放自从上一次启动以来的所有事件，然后成功传入信息。

要注意的是，在编写本书时，Akka Persistence 模块仍然只被标记为实验版本（本书最后一次编辑完成前几个礼拜，Akka 2.4 发布了正式版本）。被标记为实验版本的模块并不保证小版本之间的二进制兼容性。因此一定要理解所使用版本的情况在小心地将其添加到项目之中。可以从 Akka 文档中查阅更多关于 Akka Persistence 的信息。

9.1.5　Akka I/O

Akka I/O 是在 Akka 2.2 中引入的，Akka 团队和 Spray 团队基于 Spray 的底层 I/O 模块共同努力实现了 Akka I/O。该模块提供了一些底层 TCP 以及套接字抽象，用于编写自定义的网络通信。由于 Akka 本身试图提高抽象的层次，所以本书重点把 Akka 作为一个为用户解决了网络通信问题的工具集来介绍。但是如果想要自己处理 TCP 通信的话，Akka 也提供了相应的工具帮助我们完成工作。

自己处理 TCP 通信是一个相当高级的话题，超出了本书的范围，但是读者可以阅读一些库（比如 Brando Redis 客户端库），这些库给出了一些很好的使用 Akka I/O 来处理 TCP 通信的例子。可以从 https://github.com/chrisdinn/brando 访问该库。Redis 的通信比较简单，是一个用来分析的好例子。

9.1.6　Akka Streams 与 HTTP

在讨论邮箱相关的章节中，我们简单介绍过 Akka Streams。Streams 是 Akka 中比较有趣的新模块之一，它是 Reactive Streams 的一个实现。Akka HTTP 就是在 Spray 团队的帮助下基于 Akka Streams 实现的。

通常来说，Akka Streams 和 Reactive Streams 是更进阶的话题。如果想要研究 Akka Streams 的话，我推荐先学习 Reactive Streams。由于 Akka HTTP 是基于 Akka Streams 的，所以在学习 Akka HTTP 之前可能需要先学习 Akka Streams。正如其名，Akka HTTP 提供了编写 HTTP 客户端和服务器应用程序的抽象。

9.2　部署工具

最后要介绍的一个话题就是应用程序的部署。由于本书的目标读者是开发者，所以没有从基础设施的角度介绍应用程序的部署。

因为 Akka Cluster 可以用于应用程序之间的相互通信，所以作为开发者基本上只要在应用程序中构建好集群就行了。然而，在基础设施层面上构建新的应用程序节点可能会相当复杂。

除了本书之外，推荐读者至少了解一下比如 Puppet、Chef、Salt 和 Ansibe 这些工具——不要等到要真正部署程序了再开始考虑处理这些问题。在 DEV 和 QA 环境下实施自动化部署。这些工具的问题是都太过于面向特定的服务器。因此类似 Mesos 和 ConductR 这样的工具可能会更好，它们将底层的多节点抽象封装，对用户不可见。这些工具可以让我们把多个节点看做一个数据中心，类似一个资源池，而不是独立的实体。无论采用哪种方法，有一个基本的准则：一旦 DevOps 策略比较成熟之后，任何人都不再需要登录到服务器上去。

这是一块很大的领域，而且更多的是运维领域里面的特定知识。但是要完成自动部署，并不一定需要一个专门的 DevOps 工程师。只要作为开发者/工程师和运维团队一起合作，慢慢学习相关知识就行了。可以成立一个小组，每周聚会几个小时，讨论如何在 DEV 和 QA 环境下完成自动化部署。一旦了解了一般的自动化实践方案，就可以开始思考如何在生产部署环境下向集群中添加新节点。如果能够解决这个问题，那么将会给团队带来巨大的便利。

下面是一个简单的例子，说明 ConductR 所能提供的功能。它可以把基础设施和应用程序打包，并且可以在整个基础设施上对打包进行备份，如图 9-1 所示。

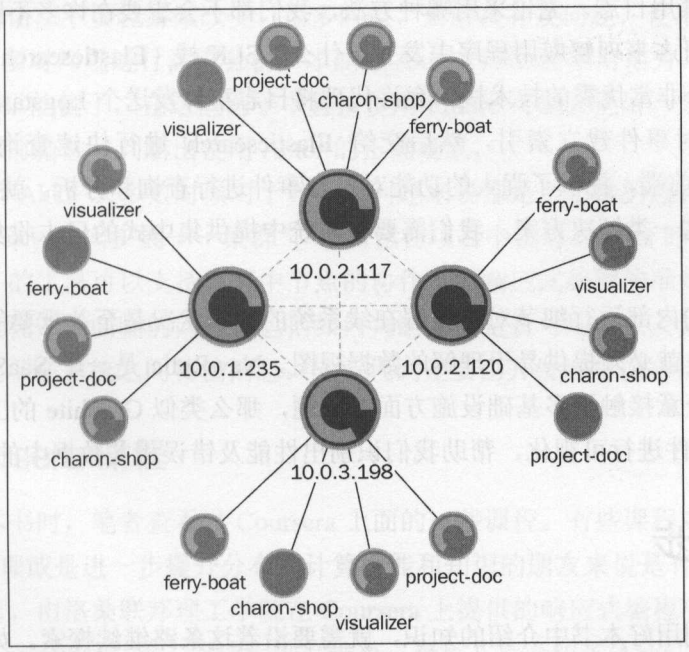


图 9-1

如果想要理解使用传统方法部署现代应用程序的问题，可以查阅 Typesafe 提供的

ConductR 白皮书。这个领域非常活跃，而且 Typesafe 提供了很多工具和方法，用于将应用程序集成到 Typesafe 平台上。

9.3　监控日志与事件

最后，在讨论完部署之后，我们需要快速地介绍一下如何监控应用程序。要确保应用程序易于维护，大概需要考虑下面三个方面：

- **事件与性能**：成功与失败的次数、网络流量以及延迟。
- **日志**：应用程序的内部行为以及失败的细节。
- **健康**：程序是否处于运行中，是否健康，还有运行效果是否在意料之中？

对于一般的时间和性能测量，可以使用类似 Statsd 或是 Metrics 这样的工具，在应用程序内部添加计数器或是计时器，然后把统计到的数据传给另一个负责生成图标的服务。AppDynamics 和 NewRelic 是 Typesafe 的合作伙伴，它们提供了解决类似问题的 SaaS 解决方案，如果不想自己来处理这些事情的话，可以选用这些方案。

大规模的日志处理起来是非常困难的，我也见到过一些大型系统把日志级别设置成 Error，甚至是禁用日志。无论采用哪种方法，我们都不会想要在许多不同服务器上的多个文件中搜索日志来理解应用程序中发生了什么。ELK 栈（Elasticsearch，Logstash 以及 Kibana）是一个非常优秀的技术栈，允许代理将日志事件发送个 Logstash 服务集群，由 Logstash 负责对事件建立索引，然后交给 Elasticsearch 进行快速查询。Kibana 作为 Elasticsearch 的前端，提供了强大的功能对日志事件进行查询和分析。就算不使用 ELK，我也绝对推荐这一类解决方案。我们需要在系统中提供集中式的日志收集机制，提供系统的可维护性。

查阅系统的内部运行细节对于理解在线系统的健康状况是至关重要的。而要能够对事件进行分析，就必须提供易于理解的数据视图。NewRelic 是一个 SaaS 模型，很容易上手。如果不介意接触更多基础设施方面的问题，那么类似 Graphite 的工具可以对使用 Statsd 收集的事件进行可视化，帮助我们识别出性能及错误相关数据中的趋势[①]。

9.4　下一步

要真正地利用好本书中介绍的知识，就需要沿着这条路继续探索。如果已经在阅读本书的话，作者认为读者是希望学习分布式相关的知识的。在这一小节中，将提供更多

① 译者注：原文此处说有一张截图，但是并没有。

读者可能会感兴趣的除了 Akka 文档之外的活动以及资源，它们会对读者理解其他相关领域有所帮助。

9.4.1 编写一些 Actor 代码

现在，如果读者还没有完成每章后面的课后作业，强烈推荐回过头去看一下并试图完成这些任务。

尤其是最开始的 4 个章节（到“第 4 章：Actor 的生命周期”为止）就足够我们花上一些时间了。首先使用 Akka 来编写一个本地的并发应用程序。这是最终要的基础工作，后面的内容都是在此基础上构建的。

“第 5 章：纵向扩展”以及后面的章节讨论的是更高阶的话题，也是最有意思的部分。笔者曾经参与过许多使用 Akka 的超大型系统，但是也只使用过其中一小部分技术，原因在于有些模块（比如 Cluster）是最新研发出来的，而一些使用场景则不需要用到某些功能（比如修改邮箱配置）。

尽管本书基本上都把 Akka 作为一个分布式工具集来讨论介绍，但是它也可以同时作为一个并发框架来使用。在理解何时使用哪种方案的时候一定要优先考虑简单性，没有很好的理由的话，不要轻易做改变。很多时候，要解决并发问题，使用 Future 就已经足够了。因此，要对问题进行深度分析，才能把 Akka 的 Actor 用对地方。笔者也见过一些错误使用 Actor 的例子，在这些例子中直接使用 Future 可能要更好。只有通过实践，才能习得更好的前瞻性，判断出使用 Actor 的正确场景。

当然，尽管 Akka 大多数时候对于分布式问题来说都是个不错的选择，笔者还是绝对会学习“第 6 章：横向扩展——集群”，令自己在项目中使用 Cluster 时能够更加自信。类似 Zookeeper 的工具可以支持集群中节点的协作，而响应式编程的准则广泛涉及了系统协作与交互过程中要面临的问题，包括如何判断节点是否可用，以及如何在不增加代码复杂度的前提下在系统之间传输消息。毕竟代码是要由开发者自己来维护的。

9.4.2 Coursera 课程

快要完成本书时，笔者查看过 Coursera 上面的一些课程。有些课程对于想要学习更多 Akka 工作原理或是进一步提升分布式计算技能和知识的朋友来说是有所帮助的。

本书完成时，由洛桑联邦理工学院在 Coursera 上提供的响应式编程准则（Principles of Reactive Programming）第二次课程刚好结束。最后几周介绍了 Akka、Cluster 及其相关话题。对于任何想了解更多通用响应式编程方面知识的人来说，可以学习这个不错的课程。这样也可以多做一些课后作业，有助于学习。课程中提到的概念可能会与本书有

所重复，不过这自然会强化对这些话题的理解。可以从 https://www.coursera.org/course/reactive 处访问。

如果读者现在是一名 Java 开发者的话，Scala 函数式编程（Functional Programming in Scala）课程是个很好的学习函数式编程的资源。可以从 https://www.coursera.org/course/progfun 处访问。

结合本书一起学习的最佳课程之一就是伊利诺伊大学厄巴纳-香槟分校提供的云计算概念（Cloud Computing Concepts）。该课程涉及了许多 Akka 团队用来编写 Cluster 的话题。课程中详细介绍了 Gossip 协议以及用于得到集群一致性视图的时序概念（比如向量时钟）。深入理解这些概念对于我们的学习绝对很有帮助。可以从如下链接访问该课程：

- **第 1 部分**：https://www.coursera.org/course/cloudcomputing
- **第 2 部分**：https://www.coursera.org/course/cloudcomputing2

9.5　小结

本书介绍了 Akka 的使用方法，试图展示分布式系统的构建方法并介绍了一旦涉及到大规模分布式计算就会遇到的许多问题。本章是最后一章，列出了读者应该知道的一些 Akka 话题、一些部署问题以及下一步读者可以继续自己学习的内容。希望本书介绍的概念对于读者开始使用 Akka 能够有所帮助。Akka 是个很有趣也很强大的工具集。虽然文档介绍了 Akka 的使用方法，但是并没有总是告诉用户何时何地应该使用框架的哪个部分。希望本书覆盖了“为什么使用”和“什么时候使用”这两点，这样读者就能理解这些工具的优势及其带来的好处。

恭喜完成了本书的阅读，祝未来的学习之路好运！

Akka入门与实践

Akka是一个分布式计算工具集，支持开发者使用Java以及Scala便捷地构建正确的并发分布式应用程序。用户使用Akka构建的应用程序能够扩展至多台服务器，并能通过自恢复对失效情况做出响应。

《Akka入门与实践》旨在对Akka进行系统的介绍。对刚开始使用Akka构建并发分布式应用程序的读者来说，本书将带领大家学习所需的所有概念。本书首先从Actor的概念开始，然后循序渐进介绍了Akka中的并发实现以及网络应用程序的构建。在教授如何使用Akka来解决疑难问题的同时，本书将介绍如何规避其中常见的陷阱。

本书的读者

本书面向初级到中级的Java或Scala开发者，以帮助他们构建满足当今大规模用户需求的应用程序。如果你的应用程序需要处理日益增长的用户量以及数据集，同时又要满足高性能需求，那么这本书正是你所需要的。

从本书中学到的知识

- 使用Akka来解决并发编程中遇到的挑战
- 使用Akka来构建集群，并在多台机器上分配工作
- 扩展应用程序，使之支持大量并发用户
- 利用自恢复的应用程序为系统提供容错性
- 使用事件驱动的方法构建低延迟应用程序
- 通过高效利用系统资源来减少硬件开销
- 通过扩展来最大化网络效率

异步社区
人民邮电出版社
www.epubit.com.cn

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

美术编辑：董志桢

ISBN 978-7-115-45354-9

ISBN 978-7-115-45354-9
定价:49.00 元

分类建议：计算机／程序设计
人民邮电出版社网址：www.ptpress.com.cn